book_to_slide_BY_sections_V12

July 13, 2025

1 Set up Paths

[ ]:

```
# Cell 1: Setup and Configuration
import os
import re
import logging
import warnings
from docx import Document
import pdfplumber
import ollama
from tenacity import retry, stop_after_attempt, wait_exponential, RetryError
import json
import logging
# Setup Logger for this cell
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
  ↪%(message)s')
logger = logging.getLogger(__name__)

# --- 1. CORE SETTINGS ---
# Set this to True for EPUB, False for PDF. This controls the entire notebook's␣
  ↪flow.
PROCESS_EPUB = True # for EPUB
# PROCESS_EPUB = False # for PDF

# --- 2. INPUT FILE NAMES ---
# The name of the Unit Outline file (e.g., DOCX, PDF)
UNIT_OUTLINE_FILENAME = "ICT312 Digital Forensic_Final.docx" # epub
# UNIT_OUTLINE_FILENAME = "ICT311 Applied Cryptography.docx" # pdf

EXTRACT_UO = False

CREATE_RAG_BOOK = False

# The names of the book files
EPUB_BOOK_FILENAME = "Bill Nelson, Amelia Phillips, Christopher Steuart - Guide␣
  ↪to Computer Forensics and Investigations_ Processing Digital␣
  ↪Evidence-Cengage Learning (2018).epub"
```

```python
PDF_BOOK_FILENAME = "(Chapman & Hall_CRC Cryptography and Network Security Series) Jonathan Katz, Yehuda Lindell - Introduction to Modern Cryptography-CRC Press (2020).pdf"

# --- 3. DIRECTORY STRUCTURE ---
# Define the base path to your project to avoid hardcoding long paths everywhere
PROJECT_BASE_DIR = "/home/sebas_dev_linux/projects/course_generator"

# Define subdirectories relative to the base path
DATA_DIR = os.path.join(PROJECT_BASE_DIR, "data")
PARSE_DATA_DIR = os.path.join(PROJECT_BASE_DIR, "Parse_data")

# Construct full paths for clarity
INPUT_UO_DIR = os.path.join(DATA_DIR, "UO")
INPUT_BOOKS_DIR = os.path.join(DATA_DIR, "books")
OUTPUT_PARSED_UO_DIR = os.path.join(PARSE_DATA_DIR, "Parse_UO")
OUTPUT_PARSED_TOC_DIR = os.path.join(PARSE_DATA_DIR, "Parse_TOC_books")
OUTPUT_DB_DIR = os.path.join(DATA_DIR, "DataBase_Chroma")



# New configuration file paths
CONFIG_DIR = os.path.join(PROJECT_BASE_DIR, "configs")
SETTINGS_DECK_PATH = os.path.join(CONFIG_DIR, "settings_deck.json")
TEACHING_FLOWS_PATH = os.path.join(CONFIG_DIR, "teaching_flows.json")

# to check the layauts
LAYOUT_MAPPING_PATH = os.path.join(CONFIG_DIR, "layout_mapping.json")


# New output path for the processed settings
PROCESSED_SETTINGS_PATH = os.path.join(CONFIG_DIR, "processed_settings.json")

# to Save the individual FINAL plan to a file
PLAN_OUTPUT_DIR = os.path.join(PROJECT_BASE_DIR, "generated_plans")
os.makedirs(PLAN_OUTPUT_DIR, exist_ok=True)

#to Save the individual FINAL Content to a file
CONTENT_OUTPUT_DIR = os.path.join(PROJECT_BASE_DIR, "generated_content")
os.makedirs(CONTENT_OUTPUT_DIR, exist_ok=True)

CONTENT_LLM_OUTPUT_DIR = os.path.join(PROJECT_BASE_DIR, "generated_content_llm")
os.makedirs(CONTENT_LLM_OUTPUT_DIR, exist_ok=True)


SLIDE_TEMPLATE_PATH = "/home/sebas_dev_linux/projects/course_generator/data/slide_style/slide_style_test.pptx"
```

```
FINAL_PRESENTATION_DIR = os.path.join(PROJECT_BASE_DIR, "final_presentations")
os.makedirs(FINAL_PRESENTATION_DIR, exist_ok=True)


# --- 4. LLM & EMBEDDING CONFIGURATION ---
LLM_PROVIDER = "ollama"  # Can be "ollama", "openai", "gemini"
OLLAMA_HOST = "http://localhost:11434"
OLLAMA_MODEL = "qwen3:8b" # "qwen3:8b", #"mistral:latest"
EMBEDDING_MODEL_OLLAMA = "nomic-embed-text"
CHUNK_SIZE = 800
CHUNK_OVERLAP = 100

# --- 5. DYNAMICALLY GENERATED PATHS & IDs (DO NOT EDIT THIS SECTION) ---
# This section uses the settings above to create all the necessary variables␣
 ↪for later cells.

# Extract Unit ID from the filename
# --- Helper Functions ---
def print_header(text: str, char: str = "="):
    """Prints a centered header to the console."""
    print("\n" + char * 80)
    print(text.center(80))
    print(char * 80)

def extract_uo_id_from_filename(filename: str) -> str:
    match = re.match(r'^[A-Z]+\d+', os.path.basename(filename))
    if match:
        return match.group(0)
    raise ValueError(f"Could not extract a valid Unit ID from filename:␣
 ↪'{filename}'")

try:
    UNIT_ID = extract_uo_id_from_filename(UNIT_OUTLINE_FILENAME)
except ValueError as e:
    print(f"Error: {e}")
    UNIT_ID = "UNKNOWN_ID"

# Full path to the unit outline file
FULL_PATH_UNIT_OUTLINE = os.path.join(INPUT_UO_DIR, UNIT_OUTLINE_FILENAME)

# Determine which book and output paths to use based on the PROCESS_EPUB flag
if PROCESS_EPUB:
    BOOK_PATH = os.path.join(INPUT_BOOKS_DIR, EPUB_BOOK_FILENAME)
    PRE_EXTRACTED_TOC_JSON_PATH = os.path.join(OUTPUT_PARSED_TOC_DIR,␣
 ↪f"{UNIT_ID}_epub_table_of_contents.json")
else:
    BOOK_PATH = os.path.join(INPUT_BOOKS_DIR, PDF_BOOK_FILENAME)
    PRE_EXTRACTED_TOC_JSON_PATH = os.path.join(OUTPUT_PARSED_TOC_DIR,␣
  ↪f"{UNIT_ID}_pdf_table_of_contents.json")

# Define paths for the vector database
file_type_suffix = 'epub' if PROCESS_EPUB else 'pdf'
CHROMA_PERSIST_DIR = os.path.join(OUTPUT_DB_DIR,␣
  ↪f"chroma_db_toc_guided_chunks_{file_type_suffix}")
CHROMA_COLLECTION_NAME = f"book_toc_guided_chunks_{file_type_suffix}_v2"

# Define path for the parsed unit outline
PARSED_UO_JSON_PATH = os.path.join(OUTPUT_PARSED_UO_DIR, f"{os.path.
  ↪splitext(UNIT_OUTLINE_FILENAME)[0]}_parsed.json")

# --- Sanity Check Printout ---
print("--- CONFIGURATION SUMMARY ---")
print(f"Processing Mode: {'EPUB' if PROCESS_EPUB else 'PDF'}")
print(f"Unit ID: {UNIT_ID}")
print(f"Unit Outline Path: {FULL_PATH_UNIT_OUTLINE}")
print(f"Book Path: {BOOK_PATH}")
print(f"Parsed UO Output Path: {PARSED_UO_JSON_PATH}")
print(f"Parsed ToC Output Path: {PRE_EXTRACTED_TOC_JSON_PATH}")
print(f"Vector DB Path: {CHROMA_PERSIST_DIR}")
print(f"Vector DB Collection: {CHROMA_COLLECTION_NAME}")
print("--- SETUP COMPLETE ---")
```

2 System Prompt

[ ]:

~~~
UNIT_OUTLINE_SYSTEM_PROMPT_TEMPLATE = """
You are an expert academic assistant tasked with parsing a university unit␣
  ↪outline document and extracting key information into a structured JSON␣
  ↪format.

The input will be the raw text content of a unit outline. Your goal is to␣
  ↪identify and extract the following details and structure them precisely as␣
  ↪specified in the JSON schema below. Note: do not change any key name

**JSON Output Schema:**

```json
{{
  "unitInformation": {{
    "unitCode": "string | null",
    "unitName": "string | null",
    "creditPoints": "integer | null",
    "unitRationale": "string | null",
    "prerequisites": "string | null"
~~~

```
  }},
  "learningOutcomes": [
    "string"
  ],
  "assessments": [
    {{
      "taskName": "string",
      "description": "string",
      "dueWeek": "string | null",
      "weightingPercent": "integer | null",
      "learningOutcomesAssessed": "string | null"
    }}
  ],
  "weeklySchedule": [
    {{
      "week": "string",
      "contentTopic": "string",
      "requiredReading": "string | null"
    }}
  ],
  "requiredReadings": [
    "string"
  ],
  "recommendedReadings": [
    "string"
  ]
}}

Instructions for Extraction:
Unit Information: Locate Unit Code, Unit Name, Credit Points. Capture 'Unit␣
 ↪Overview / Rationale' as unitRationale. Identify prerequisites.
Learning Outcomes: Extract each learning outcome statement.
Assessments: Each task as an object. Capture full task name, description, Due␣
 ↪Week, Weighting % (number), and Learning Outcomes Assessed.
weeklySchedule: Each week as an object. Capture Week, contentTopic, and␣
 ↪requiredReading.
Required and Recommended Readings: List full text for each.
**Important Considerations for the LLM**:
Pay close attention to headings and table structures.
If information is missing, use null for string/integer fields, or an empty list␣
 ↪[] for array fields.
Do no change keys in the template given
Ensure the output is ONLY the JSON object, starting with {{{{ and ending with␣
 ↪}}}}. No explanations or conversational text before or after the JSON.
Now, parse the following unit outline text:
--- UNIT_OUTLINE_TEXT_START ---
{outline_text}
```

```
--- UNIT_OUTLINE_TEXT_END ---
"""
```

[ ]:

```python
# Place this in a new cell after your imports, or within Cell 3 before the functions.
# This code is based on the schema from your screenshot on page 4.

from pydantic import BaseModel, Field, ValidationError
from typing import List, Optional
import time

# Define Pydantic models that match your JSON schema
class UnitInformation(BaseModel):
    unitCode: Optional[str] = None
    unitName: Optional[str] = None
    creditPoints: Optional[int] = None
    unitRationale: Optional[str] = None
    prerequisites: Optional[str] = None

class Assessment(BaseModel):
    taskName: str
    description: str
    dueWeek: Optional[str] = None
    weightingPercent: Optional[int] = None
    learningOutcomesAssessed: Optional[str] = None

class WeeklyScheduleItem(BaseModel):
    week: str
    contentTopic: str
    requiredReading: Optional[str] = None

class ParsedUnitOutline(BaseModel):
    unitInformation: UnitInformation
    learningOutcomes: List[str]
    assessments: List[Assessment]
    weeklySchedule: List[WeeklyScheduleItem]
    requiredReadings: List[str]
    recommendedReadings: List[str]
```

2.0.1 Component: Definitive PowerPoint Layout Inspector

Component: Definitive PowerPoint Layout Inspector Primary Purpose The inspect_and_generate_layout_config function serves as a critical pre-processing utility for the automated presentation generation system. Its primary purpose is to bridge the gap between a visual PowerPoint template and the programmatic logic of the content generation agents. It achieves this by performing a deep inspection of a given PowerPoint (.pptx) template file and auto-generating a detailed, structured, and human-readable JSON configuration file (layout_mapping.json). This configuration file acts as the "API documentation" for the presentation template, allowing both human users and a Large Language Model (LLM) to understand and utilize the available slide layouts effectively. Key Functions and GoOf course. Here is a formal description of the purpose and functionality of the "Definitive PowerPoint Layout Inspector" script. This description is suitable for project documentation, a README file, or for explaining its role to other developers.

Primary Purpose The `inspect_and_generate_layout_config` function serves as a critical pre-processing utility for the automated presentation generation system. Its primary purpose is to **bridge the gap between a visual PowerPoint template and the programmatic logic of the content generation agents**.

It achieves this by performing a deep inspection of a given PowerPoint (`.pptx`) template file and auto-generating a detailed, structured, and human-readable JSON configuration file (`layout_mapping.json`). This configuration file acts as the "API documentation" for the presentation template, allowing both human users and a Large Language Model (LLM) to understand and utilize the available slide layouts effectively.

Key Functions and Goals

1. **Comprehensive Layout Discovery:**
   - The script guarantees that **every single slide layout** present in the PowerPoint template's Slide Master is detected and analyzed. This prevents a common problem where unused or unconventionally named layouts might be missed by simpler scripts.
2. **Detailed Placeholder Analysis:**
   - For each layout, the script extracts an exhaustive list of all its placeholders. For every placeholder, it records crucial metadata:
     - `type`: The functional role of the placeholder (e.g., `TITLE`, `BODY`, `OBJECT`, `TABLE`, `PICTURE`).
     - `name`: The unique name given to the placeholder in the PowerPoint interface (e.g., "Title 1", "Content Placeholder 2").
     - `idx`: The internal identification number of the placeholder.
     - `position and size`: The physical coordinates (`left`, `top`) and dimensions (`width`, `height`), converted to an intuitive unit (inches) for easy comprehension of the layout's visual structure.
3. **Intelligent Capability Summarization:**
   - The script's core innovation is its ability to generate a **machine-readable capabilities summary** for each layout. Instead of just listing raw data, it synthesizes the placeholder information into a concise description of what the layout is designed for. For example:
     - `{"title_support": "standard_title", "body_layout": "2_column"}`
     - `{"title_support": "centered_title_with_subtitle", "body_layout": "no_body"}`
     - `{"specific_content_types": ["TABLE", "CHART"]}`
   - This summary is specifically designed to be passed to an LLM as part of a prompt, enabling the LLM to make an informed, logical choice about the best layout for presenting a given piece of information.
4. **User-Friendly Configuration:**
   - While providing a detailed summary for the LLM, the script also generates a simplified `user_selections` section. This allows a human operator to easily map the system's

semantic slide types (e.g., “Agenda”, “Summary”) to a specific layout index, providing a robust fallback and manual override capability.

How It Solves Critical Problems

- **Eliminates Ambiguity:** By capturing the name, index, and position of every placeholder, it solves the problem of layouts with multiple placeholders of the same type (e.g., two content boxes). The system can now programmatically target the “left column” vs. the “right column”.
- **Decouples Logic from Design:** The presentation generation agent no longer needs hard-coded assumptions about the template’s design. All the logic for choosing and populating layouts is driven by the generated JSON file. This means the visual template can be updated or completely replaced without requiring changes to the core Python code.
- **Empowers the LLM:** It transforms a visual, unstructured design asset (the `.pptx` file) into a structured, well-defined set of “tools” (the layouts) that an LLM can reason about. This is the key to enabling more advanced tasks where the LLM doesn’t just fill in content, but also makes decisions about the *visual structure* of the presentation.

In summary, the Definitive PowerPoint Layout Inspector is the foundational component that makes the entire presentation generation process intelligent, configurable, and robust. It translates the abstract design of a template into concrete, actionable data.

[ ]:

```python
# this process the layauts from the slides provide to use them to process the plan (This require be change to the course of the system  )
# https://aistudio.google.com/prompts/18YaU5pG96eFMbM1l6yeG1v9j63PkLc5H

import logging
import os
import json
import logging
import random
from pptx import Presentation
from pptx.util import Inches
from pptx.enum.shapes import MSO_SHAPE_TYPE
from pptx.enum.shapes import PP_PLACEHOLDER
# Setup Logger for this cell
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)
def print_header(text: str, char: str = "="):
    """Prints a centered header to the console."""
    print("\n" + char * 80)
    print(text.center(80))
    print(char * 80)

def generate_layout_capabilities(layout_name: str, placeholders: list) -> dict:

    essential_placeholders = [p for p in placeholders if p['type'] != 'SLIDE_NUMBER']
    capabilities = {"title_support": "none", "body_layout": "none", "specific_content_types": []}
    has_center_title = any(p['type'] == 'CENTER_TITLE' for p in essential_placeholders)
    has_standard_title = any(p['type'] == 'TITLE' for p in essential_placeholders)
    has_subtitle = any(p['type'] == 'SUBTITLE' for p in essential_placeholders)
    if has_center_title: capabilities["title_support"] = "centered_title"
    elif has_standard_title: capabilities["title_support"] = "standard_title"
    if has_subtitle: capabilities["title_support"] += "_with_subtitle"
    body_placeholders = [p for p in essential_placeholders if p['type'] not in ('TITLE', 'CENTER_TITLE', 'SUBTITLE')]
    if len(body_placeholders) == 0: capabilities["body_layout"] = "no_body"
    elif len(body_placeholders) == 1: capabilities["body_layout"] = "single_column"
    elif len(body_placeholders) > 1:
        body_placeholders.sort(key=lambda p: p['left'])
        if len(body_placeholders) > 1 and body_placeholders[1]['left'] > (body_placeholders[0]['left'] + body_placeholders[0]['width'] * 0.5):
            capabilities["body_layout"] = f"{len(body_placeholders)}_column"
        else: capabilities["body_layout"] = "stacked_sections"
    specific_types = {p['type'] for p in body_placeholders if p['type'] not in ('BODY', 'OBJECT')}
    if specific_types: capabilities["specific_content_types"] = sorted(list(specific_types))
    return capabilities


def inspect_and_generate_layout_config(template_path: str, output_path: str):
    """
    Inspects a template, generates a machine-readable capabilities summary, and creates
    a complete and definitive JSON configuration file for user and LLM use.
    """
    try:
        prs = Presentation(template_path)
    except Exception as e:
        logger.error(f"Could not open or parse the presentation template at '{template_path}'. Error: {e}")
        return

    # --- 1. Analyze all Layouts ---

    available_layouts = []
    for i, layout in enumerate(prs.slide_layouts):
        placeholder_details = []
        for p in layout.placeholders:
            placeholder_details.append({
                "idx": p.placeholder_format.idx, "type": p.placeholder_format.
↪type.name,
                "name": p.name, "left": round(p.left.inches, 2), "top": round(p.
↪top.inches, 2),
                "width": round(p.width.inches, 2), "height": round(p.height.
↪inches, 2)
            })
        capabilities = generate_layout_capabilities(layout.name,␣
↪placeholder_details)
        layout_info = {"layout_index": i, "layout_name": layout.name,␣
↪"capabilities": capabilities, "placeholders": placeholder_details}
        available_layouts.append(layout_info)

    # --- 2. Create the Complete User-Facing Configuration Structure ---
    def find_default_by_capability(layouts, capability_key, capability_value,␣
↪fallback_index=1):
        for layout in layouts:
            if layout['capabilities'].get(capability_key) == capability_value:
                return layout['layout_index']
        # If no perfect match is found, return a safe fallback index
        return fallback_index if len(layouts) > fallback_index else 0

    # ** This map is now complete, explicitly including all required keys **
    user_selection_map = {
        "Title": { # title and subtitle
            "description": "Main title slide of the deck.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "title_support",␣
↪"centered_title_with_subtitle")
        },
        "Agenda": { # title and object
            "description": "Agenda/Table of Contents slide.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "single_column")
        },
        "Content": { # title and object
            "description": "Default slide for a standard topic with bullet␣
↪points.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "single_column")
        },
        "Content_Two_Column": { # title and 2 objects
            "description": "Slide for side-by-side content, like comparisons.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "2_column")
        },
        "Content_child": { ## # title, subtitle and object
            "description": "Default slide for a standard topic with bullet␣
↪points.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "single_column")
        },
        "Content_Two_Column_child": { ## title, subtitle and 2 objects
            "description": "Slide for side-by-side content, like comparisons.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "2_column")
        },
        "Application": {# title and  object
            "description": "Slide for interactive questions ('Let's Apply This!
↪').",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "single_column")
        },
        "Application_Two_Column ": { # title and 2 objects
            "description": "Slide for side-by-side content ('Let's Apply This!
↪').",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "2_column")
        },
        "Summary": { # title and object
            "description": "The final summary slide of the deck.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "body_layout", "single_column")
        },
        "End": { # title
            "description": "The final 'Thank You / Questions?' slide.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "title_support",␣
↪"centered_title", fallback_index=7)
        },
        "Divider": { # title and object
            "description": "Slide that divide sections general use base on the␣
↪agenda.",
            "selected_layout_index":␣
↪find_default_by_capability(available_layouts, "title_support",␣
↪"centered_title", fallback_index=7)
        }
    }
```

```python
    config_to_save = {
        "//_INSTRUCTIONS": "This file describes the available slide layouts. The 'available_layouts' section is for an LLM to read. The 'user_selections' section is for you to edit. Please verify the 'selected_layout_index' for each slide type.",
        "template_file": os.path.basename(template_path),
        "user_selections": user_selection_map,
        "available_layouts": available_layouts
    }

    # --- 3. Save the Configuration File ---
    try:
        with open(output_path, 'w', encoding='utf-8') as f:
            json.dump(config_to_save, f, indent=4)
        print_header("Configuration Generated Successfully", char="=")
        print(f"A new, human-readable configuration file has been saved to:\n{output_path}")
        print("\nPlease open this file to review the classifications and edit your layout selections.")
    except Exception as e:
        logger.error(f"Failed to write the layout configuration file to '{output_path}'. Error: {e}")

# --- Execution ---
SLIDE_TEMPLATE_PATH = "/home/sebas_dev_linux/projects/course_generator/data/slide_style/slide_style_test_2.pptx"
LAYOUT_MAPPING_PATH = "/home/sebas_dev_linux/projects/course_generator/configs/layout_mapping_test.json"
# You run this function to generate the config file.
# Make sure SLIDE_TEMPLATE_PATH and LAYOUT_MAPPING_PATH are defined.
inspect_and_generate_layout_config(SLIDE_TEMPLATE_PATH, LAYOUT_MAPPING_PATH)
```

```
================================================================================
                      Configuration Generated Successfully
================================================================================
A new, human-readable configuration file has been saved to:
/home/sebas_dev_linux/projects/course_generator/configs/layout_mapping_test.json

Please open this file to review the classifications and edit your layout
selections.
```

3 Extrac Unit outline details to process following steps - output raw json with UO details

[ ]:

```
# Cell 3: Parse Unit Outline


# --- Helper Functions for Parsing ---
def extract_text_from_file(filepath: str) -> str:
    _, ext = os.path.splitext(filepath.lower())
    if ext == '.docx':
        doc = Document(filepath)
        full_text = [p.text for p in doc.paragraphs]
        for table in doc.tables:
            for row in table.rows:
                full_text.append(" | ".join(cell.text for cell in row.cells))
        return '\n'.join(full_text)
    elif ext == '.pdf':
        with pdfplumber.open(filepath) as pdf:
            return "\n".join(page.extract_text() for page in pdf.pages if page.
 ↪extract_text())
    else:
        raise TypeError(f"Unsupported file type: {ext}")

def parse_llm_json_output(content: str) -> dict:
    try:
        match = re.search(r'\{.*\}', content, re.DOTALL)
        if not match: return None
        return json.loads(match.group(0))
    except (json.JSONDecodeError, TypeError):
        return None

@retry(stop=stop_after_attempt(3), wait=wait_exponential(min=2, max=10))
def call_ollama_with_retry(client, prompt):
    logger.info(f"Calling Ollama model '{OLLAMA_MODEL}'...")
    response = client.chat(
        model=OLLAMA_MODEL,
        messages=[{"role": "user", "content": prompt}],
        format="json",
        options={"temperature": 0.0}
    )
    if not response or 'message' not in response or not response['message'].
 ↪get('content'):
        raise ValueError("Ollama returned an empty or invalid response.")
    return response['message']['content']

# --- Main Orchestration Function for this Cell ---
def parse_and_save_outline_robust(
```

```python
    input_filepath: str,
    output_filepath: str,
    prompt_template: str,
    max_retries: int = 3
):
    logger.info(f"Starting to robustly process Unit Outline: {input_filepath}")

    if not os.path.exists(input_filepath):
        logger.error(f"Input file not found: {input_filepath}")
        return

    try:
        outline_text = extract_text_from_file(input_filepath)
        if not outline_text.strip():
            logger.error("Extracted text is empty. Aborting.")
            return
    except Exception as e:
        logger.error(f"Failed to extract text from file: {e}", exc_info=True)
        return

    client = ollama.Client(host=OLLAMA_HOST)
    current_prompt = prompt_template.format(outline_text=outline_text)

    for attempt in range(max_retries):
        logger.info(f"Attempt {attempt + 1}/{max_retries} to parse outline.")

        try:
            # Call the LLM
            llm_output_str = call_ollama_with_retry(client, current_prompt)

            # Find the JSON blob in the response
            json_blob = parse_llm_json_output(llm_output_str) # Your existing helper
            if not json_blob:
                raise ValueError("LLM did not return a parsable JSON object.")

            # *** THE KEY VALIDATION STEP ***
            # Try to parse the dictionary into your Pydantic model.
            # This will raise a `ValidationError` if keys are wrong, types are wrong, or fields are missing.
            parsed_data = ParsedUnitOutline.model_validate(json_blob)

            # If successful, save the validated data and exit the loop
            logger.info("Successfully validated JSON structure against Pydantic model.")
            os.makedirs(os.path.dirname(output_filepath), exist_ok=True)
            with open(output_filepath, 'w', encoding='utf-8') as f:
                # Use .model_dump_json() for clean, validated output
                f.write(parsed_data.model_dump_json(indent=2))

            logger.info(f"Successfully parsed and saved Unit Outline to:␣
 ↪{output_filepath}")
            return # Exit function on success

        except ValidationError as e:
            logger.warning(f"Validation failed on attempt {attempt + 1}. Error:␣
 ↪{e}")
            # Formulate a new prompt with the error message for self-correction
            error_feedback = (
                f"\n\nYour previous attempt failed. You MUST correct the␣
 ↪following errors:\n"
                f"{e}\n\n"
                f"Please regenerate the entire JSON object, ensuring it␣
 ↪strictly adheres to the schema "
                f"and corrects these specific errors. Do not change any key␣
 ↪names."
            )
            current_prompt = current_prompt + error_feedback # Append the error␣
 ↪to the prompt

        except Exception as e:
            # Catch other errors like network issues from call_ollama_with_retry
            logger.error(f"An unexpected error occurred on attempt {attempt +␣
 ↪1}: {e}", exc_info=True)
            # You might want to wait before retrying for non-validation errors
            time.sleep(5)

    logger.error(f"Failed to get valid structured data from the LLM after␣
 ↪{max_retries} attempts.")


# --- In your execution block, call the new function ---
# parse_and_save_outline(...) becomes:

if EXTRACT_UO:
    parse_and_save_outline_robust(
        input_filepath=FULL_PATH_UNIT_OUTLINE,
        output_filepath=PARSED_UO_JSON_PATH,
        prompt_template=UNIT_OUTLINE_SYSTEM_PROMPT_TEMPLATE
    )
```

4 Extract TOC from epub or PDF

[ ]:

```
# Cell 4: Extract Book Table of Contents (ToC) with Pre-assigned IDs & Links in␣
 ↪Order

from ebooklib import epub, ITEM_NAVIGATION
from bs4 import BeautifulSoup
import fitz  # PyMuPDF
import json
import os
from typing import List, Dict
import urllib.parse # Needed to clean up links

# ================================================================================
# 1. HELPER FUNCTIONS (MODIFIED TO INCLUDE ID ASSIGNMENT AND LINK EXTRACTION)
# ================================================================================

def clean_epub_href(href: str) -> str:
    """Removes URL fragments and decodes URL-encoded characters."""
    if not href: return ""
    # Remove fragment identifier (e.g., '#section1')
    cleaned_href = href.split('#')[0]
    # Decode any URL-encoded characters (e.g., %20 -> space)
    return urllib.parse.unquote(cleaned_href)

# --- EPUB Extraction Logic ---
def parse_navpoint(navpoint: BeautifulSoup, counter: List[int], level: int = 0)␣
 ↪-> Dict:
    """Recursively parses EPUB 2 navPoints and assigns a toc_id and␣
 ↪link_filename."""
    title = navpoint.navLabel.text.strip()
    if not title: return None

    # --- MODIFICATION: Extract the linked filename ---
    content_tag = navpoint.find('content', recursive=False)
    link_filename = clean_epub_href(content_tag['src']) if content_tag else ""

    node = {
        "level": level,
        "toc_id": counter[0],
        "title": title,
        "link_filename": link_filename, # Add the cleaned link
        "children": []
    }
    counter[0] += 1

    for child_navpoint in navpoint.find_all('navPoint', recursive=False):
        child_node = parse_navpoint(child_navpoint, counter, level + 1)
        if child_node: node["children"].append(child_node)

    return node

def parse_li(li_element: BeautifulSoup, counter: List[int], level: int = 0) ->␣
 ↪Dict:
    """Recursively parses EPUB 3 <li> elements and assigns a toc_id and␣
 ↪link_filename."""
    a_tag = li_element.find('a', recursive=False)
    if a_tag:
        title = a_tag.get_text(strip=True)
        if not title: return None

        # --- MODIFICATION: Extract the linked filename ---
        link_filename = clean_epub_href(a_tag.get('href'))

        node = {
            "level": level,
            "toc_id": counter[0],
            "title": title,
            "link_filename": link_filename, # Add the cleaned link
            "children": []
        }
        counter[0] += 1

        nested_ol = li_element.find('ol', recursive=False)
        if nested_ol:
            for sub_li in nested_ol.find_all('li', recursive=False):
                child_node = parse_li(sub_li, counter, level + 1)
                if child_node: node["children"].append(child_node)
        return node
    return None

def extract_epub_toc(epub_path, output_json_path):
    print(f"Processing EPUB ToC for: {epub_path}")
    toc_data = []
    book = epub.read_epub(epub_path)
    id_counter = [0]

    for nav_item in book.get_items_of_type(ITEM_NAVIGATION):
        soup = BeautifulSoup(nav_item.get_content(), 'xml')
        # Logic to handle both EPUB 2 (NCX) and EPUB 3 (XHTML)
        if nav_item.get_name().endswith('.ncx'):
            print("INFO: Found EPUB 2 (NCX) Table of Contents. Parsing...")
            navmap = soup.find('navMap')
            if navmap:
                for navpoint in navmap.find_all('navPoint', recursive=False):
                    node = parse_navpoint(navpoint, id_counter, level=0)
                    if node: toc_data.append(node)
        else: # Assumes EPUB 3
            print("INFO: Found EPUB 3 (XHTML) Table of Contents. Parsing...")
            toc_nav = soup.select_one('nav[epub|type="toc"]')
            if toc_nav:
                top_ol = toc_nav.find('ol', recursive=False)
                if top_ol:
                    for li in top_ol.find_all('li', recursive=False):
                        node = parse_li(li, id_counter, level=0)
                        if node: toc_data.append(node)
        if toc_data: break

    if toc_data:
        os.makedirs(os.path.dirname(output_json_path), exist_ok=True)
        with open(output_json_path, 'w', encoding='utf-8') as f:
            json.dump(toc_data, f, indent=2, ensure_ascii=False)
        print(f"  Successfully wrote EPUB ToC with IDs and links to:␣
 ↪{output_json_path}")
    else:
        print("  WARNING: No ToC data extracted from EPUB.")

# --- PDF Extraction Logic (Unchanged) ---
def build_pdf_hierarchy_with_ids(toc_list: List) -> List[Dict]:
    root = []
    parent_stack = {-1: {"children": root}}
    id_counter = [0]
    for level, title, page in toc_list:
        normalized_level = level - 1
        node = {"level": normalized_level, "toc_id": id_counter[0], "title":␣
 ↪title.strip(), "page": page, "children": []}
        id_counter[0] += 1
        parent_node = parent_stack.get(normalized_level - 1)
        if parent_node: parent_node["children"].append(node)
        parent_stack[normalized_level] = node
    return root

def extract_pdf_toc(pdf_path, output_json_path):
    print(f"Processing PDF ToC for: {pdf_path}")
    try:
        doc = fitz.open(pdf_path)
        toc = doc.get_toc()
        hierarchical_toc = []
        if not toc: print("  WARNING: This PDF has no embedded bookmarks (ToC).
 ↪")
        else:
            print(f"INFO: Found {len(toc)} bookmark entries. Building hierarchy␣
 ↪and assigning IDs...")
            hierarchical_toc = build_pdf_hierarchy_with_ids(toc)
        os.makedirs(os.path.dirname(output_json_path), exist_ok=True)
        with open(output_json_path, 'w', encoding='utf-8') as f:
            json.dump(hierarchical_toc, f, indent=2, ensure_ascii=False)
        print(f"  Successfully wrote PDF ToC with assigned IDs to:␣
 ↪{output_json_path}")
    except Exception as e: print(f"An error occurred during PDF ToC extraction:␣
 ↪{e}")

# ==============================================================================
# 2. EXECUTION BLOCK
# ==============================================================================
if PROCESS_EPUB:
    extract_epub_toc(BOOK_PATH, PRE_EXTRACTED_TOC_JSON_PATH)
else:
    extract_pdf_toc(BOOK_PATH, PRE_EXTRACTED_TOC_JSON_PATH)
```

5 Hirachical DB base on TOC

5.1 Process Book

[ ]:

```
# Cell 5: Create Hierarchical Vector Database (with Sequential ToC ID and Chunk␣
 ↪ID)
# This cell processes the book, enriches it with hierarchical and sequential␣
 ↪metadata,
# chunks it, and creates the final vector database.

import os
import json
import shutil
import logging
from typing import List, Dict, Any, Tuple
from langchain_core.documents import Document
from langchain_community.document_loaders import PyPDFLoader,␣
 ↪UnstructuredEPubLoader
from langchain_ollama.embeddings import OllamaEmbeddings
from langchain_chroma import Chroma
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Setup Logger for this cell
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
 ↪%(message)s')
logger = logging.getLogger(__name__)
```

```python
# --- Helper: Clean metadata values for ChromaDB ---
def clean_metadata_for_chroma(value: Any) -> Any:
    """Sanitizes metadata values to be compatible with ChromaDB."""
    if isinstance(value, list): return ", ".join(map(str, value))
    if isinstance(value, dict): return json.dumps(value)
    if isinstance(value, (str, int, float, bool)) or value is None: return value
    return str(value)

# --- Core Function to Process Book with Pre-extracted ToC ---
def process_book_with_extracted_toc(
    book_path: str,
    extracted_toc_json_path: str,
    chunk_size: int,
    chunk_overlap: int
) -> Tuple[List[Document], List[Dict[str, Any]]]:

    logger.info(f"Processing book '{os.path.basename(book_path)}' using ToC␣
 ↪from '{os.path.basename(extracted_toc_json_path)}'.")

    # 1. Load the pre-extracted hierarchical ToC
    try:
        with open(extracted_toc_json_path, 'r', encoding='utf-8') as f:
            hierarchical_toc = json.load(f)
        if not hierarchical_toc:
            logger.error(f"Pre-extracted ToC at '{extracted_toc_json_path}' is␣
 ↪empty or invalid.")
            return [], []
        logger.info(f"Successfully loaded pre-extracted ToC with␣
 ↪{len(hierarchical_toc)} top-level entries.")
    except Exception as e:
        logger.error(f"Error loading pre-extracted ToC JSON: {e}",␣
 ↪exc_info=True)
        return [], []

    # 2. Load all text elements/pages from the book
    all_raw_book_docs: List[Document] = []
    _, file_extension = os.path.splitext(book_path.lower())

    if file_extension == ".epub":
        loader = UnstructuredEPubLoader(book_path, mode="elements",␣
 ↪strategy="fast")
        try:
            all_raw_book_docs = loader.load()
            logger.info(f"Loaded {len(all_raw_book_docs)} text elements from␣
 ↪EPUB.")
        except Exception as e:
            logger.error(f"Error loading EPUB content: {e}", exc_info=True)
            return [], hierarchical_toc
    elif file_extension == ".pdf":
        loader = PyPDFLoader(book_path)
        try:
            all_raw_book_docs = loader.load()
            logger.info(f"Loaded {len(all_raw_book_docs)} pages from PDF.")
        except Exception as e:
            logger.error(f"Error loading PDF content: {e}", exc_info=True)
            return [], hierarchical_toc
    else:
        logger.error(f"Unsupported book file format: {file_extension}")
        return [], hierarchical_toc

    if not all_raw_book_docs:
        logger.error("No text elements/pages loaded from the book.")
        return [], hierarchical_toc

    # 3. Create enriched LangChain Documents by matching ToC to content
    final_documents_with_metadata: List[Document] = []

    # Flatten the ToC, AND add a unique sequential ID for sorting and␣
 ↪validation.
    flat_toc_entries: List[Dict[str, Any]] = []

    def _add_ids_and_flatten_recursive(nodes: List[Dict[str, Any]],␣
 ↪current_titles_path: List[str], counter: List[int]):
        """
        Recursively traverses ToC nodes to flatten them and assign a unique,␣
 ↪sequential toc_id.
        """
        for node in nodes:
            toc_id = counter[0]
            counter[0] += 1
            title = node.get("title", "").strip()
            if not title: continue
            new_titles_path = current_titles_path + [title]
            entry = {
                "titles_path": new_titles_path,
                "level": node.get("level"),
                "full_title_for_matching": title,
                "toc_id": toc_id
            }
            if "page" in node: entry["page"] = node["page"]
            flat_toc_entries.append(entry)
            if node.get("children"):
                _add_ids_and_flatten_recursive(node.get("children", []), new_titles_path, counter)

    toc_id_counter = [0]
    _add_ids_and_flatten_recursive(hierarchical_toc, [], toc_id_counter)
    logger.info(f"Flattened ToC and assigned sequential IDs to {len(flat_toc_entries)} entries.")

    # Logic for PDF metadata assignment
    if file_extension == ".pdf" and any("page" in entry for entry in flat_toc_entries):
        logger.info("Assigning metadata to PDF pages based on ToC page numbers...")
        flat_toc_entries.sort(key=lambda x: x.get("page", -1) if x.get("page") is not None else -1)
        for page_doc in all_raw_book_docs:
            page_num_0_indexed = page_doc.metadata.get("page", -1)
            page_num_1_indexed = page_num_0_indexed + 1
            assigned_metadata = {"source": os.path.basename(book_path), "page_number": page_num_1_indexed}
            best_match_toc_entry = None
            for toc_entry in flat_toc_entries:
                toc_page = toc_entry.get("page")
                if toc_page is not None and toc_page <= page_num_1_indexed:
                    if best_match_toc_entry is None or toc_page > best_match_toc_entry.get("page", -1):
                        best_match_toc_entry = toc_entry
                elif toc_page is not None and toc_page > page_num_1_indexed:
                    break
            if best_match_toc_entry:
                for i, title_in_path in enumerate(best_match_toc_entry["titles_path"]):
                    assigned_metadata[f"level_{i+1}_title"] = title_in_path
                assigned_metadata['toc_id'] = best_match_toc_entry.get('toc_id')
            else:
                assigned_metadata["level_1_title"] = "Uncategorized PDF Page"
            cleaned_meta = {k: clean_metadata_for_chroma(v) for k, v in assigned_metadata.items()}
            final_documents_with_metadata.append(Document(page_content=page_doc.page_content, metadata=cleaned_meta))

    # Logic for EPUB metadata assignment
    elif file_extension == ".epub":
        logger.info("Assigning metadata to EPUB elements by matching ToC titles in text...")
        toc_titles_for_search = [entry for entry in flat_toc_entries if entry.
↪get("full_title_for_matching")]
        current_hierarchy_metadata = {}
        for element_doc in all_raw_book_docs:
            element_text = element_doc.page_content.strip() if element_doc.
↪page_content else ""
            if not element_text: continue
            for toc_entry in toc_titles_for_search:
                if element_text == toc_entry["full_title_for_matching"]:
                    current_hierarchy_metadata = {"source": os.path.
↪basename(book_path)}
                    for i, title_in_path in enumerate(toc_entry["titles_path"]):
                        current_hierarchy_metadata[f"level_{i+1}_title"] =␣
↪title_in_path
                    current_hierarchy_metadata['toc_id'] = toc_entry.
↪get('toc_id')
                    if "page" in toc_entry:␣
↪current_hierarchy_metadata["epub_toc_page"] = toc_entry["page"]
                    break
            if not current_hierarchy_metadata:
                doc_metadata_to_assign = {"source": os.path.
↪basename(book_path), "level_1_title": "EPUB Preamble", "toc_id": -1}
            else:
                doc_metadata_to_assign = current_hierarchy_metadata.copy()
            cleaned_meta = {k: clean_metadata_for_chroma(v) for k, v in␣
↪doc_metadata_to_assign.items()}
            final_documents_with_metadata.
↪append(Document(page_content=element_text, metadata=cleaned_meta))

    else: # Fallback
        final_documents_with_metadata = all_raw_book_docs

    if not final_documents_with_metadata:
        logger.error("No documents were processed or enriched with hierarchical␣
↪metadata.")
        return [], hierarchical_toc

    logger.info(f"Total documents prepared for chunking:␣
↪{len(final_documents_with_metadata)}")

    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        length_function=len
    )
    final_chunks = text_splitter.split_documents(final_documents_with_metadata)
```

```python
    logger.info(f"Split into {len(final_chunks)} final chunks, inheriting hierarchical metadata.")

    # --- MODIFICATION START: Add a unique, sequential chunk_id to each chunk ---
    logger.info("Assigning sequential chunk_id to all final chunks...")
    for i, chunk in enumerate(final_chunks):
        chunk.metadata['chunk_id'] = i
    logger.info(f"Assigned chunk_ids from 0 to {len(final_chunks) - 1}.")
    # --- MODIFICATION END ---

    return final_chunks, hierarchical_toc

# --- Main Execution Block for this Cell ---
if CREATE_RAG_BOOK:
    if not os.path.exists(PRE_EXTRACTED_TOC_JSON_PATH):
        logger.error(f"CRITICAL: Pre-extracted ToC file not found at '{PRE_EXTRACTED_TOC_JSON_PATH}'.")
        logger.error("Please run the 'Extract Book Table of Contents (ToC)' cell (Cell 4) first.")
    else:
        final_chunks_for_db, toc_reloaded = process_book_with_extracted_toc(
            book_path=BOOK_PATH,
            extracted_toc_json_path=PRE_EXTRACTED_TOC_JSON_PATH,
            chunk_size=CHUNK_SIZE,
            chunk_overlap=CHUNK_OVERLAP
        )

        if final_chunks_for_db:
            if os.path.exists(CHROMA_PERSIST_DIR):
                logger.warning(f"Deleting existing ChromaDB directory: {CHROMA_PERSIST_DIR}")
                shutil.rmtree(CHROMA_PERSIST_DIR)

            logger.info(f"Initializing embedding model '{EMBEDDING_MODEL_OLLAMA}' and creating new vector database...")
            embedding_model = OllamaEmbeddings(model=EMBEDDING_MODEL_OLLAMA)

            vector_db = Chroma.from_documents(
                documents=final_chunks_for_db,
                embedding=embedding_model,
                persist_directory=CHROMA_PERSIST_DIR,
                collection_name=CHROMA_COLLECTION_NAME
            )
            reloaded_db = Chroma(persist_directory=CHROMA_PERSIST_DIR,␣
↪embedding_function=embedding_model, collection_name=CHROMA_COLLECTION_NAME)
            count = reloaded_db._collection.count()

            print("-" * 50)
            logger.info(f" Vector DB created successfully at:␣
↪{CHROMA_PERSIST_DIR}")
            logger.info(f" Collection '{CHROMA_COLLECTION_NAME}' contains␣
↪{count} documents.")
            print("-" * 50)
        else:
            logger.error(" Failed to generate chunks. Vector DB not created.")
```

[ ]:

```
# Cell 5a: Inspecting EPUB Documents and Metadata BEFORE Chunking

import json
import os
import logging
from langchain_community.document_loaders import UnstructuredEPubLoader
from langchain_core.documents import Document

# --- Setup Logger for this inspection cell ---
logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
↪%(message)s')

def inspect_epub_preprocessing():
    """
    This function replicates the pre-chunking logic from Cell 5 for EPUB files
    to show the list of large documents with their assigned ToC metadata.
    """
    if not PROCESS_EPUB:
        print("This inspection cell is for EPUB processing. Please set␣
↪PROCESS_EPUB = True in Cell 1.")
        return

    print_header("EPUB Pre-Processing Inspection", char="~")

    # --- 1. Load the necessary data (replicating start of Cell 5) ---
    logger.info("Loading pre-extracted ToC and raw EPUB elements...")
    try:
        with open(PRE_EXTRACTED_TOC_JSON_PATH, 'r', encoding='utf-8') as f:
            hierarchical_toc = json.load(f)

        loader = UnstructuredEPubLoader(BOOK_PATH, mode="elements",␣
↪strategy="fast")
        all_raw_book_docs = loader.load()
        logger.info(f"Successfully loaded {len(all_raw_book_docs)} raw text elements from the EPUB.")
    except Exception as e:
        logger.error(f"Failed to load necessary files: {e}")
        return

    # --- 2. Flatten the ToC (replicating logic from Cell 5) ---
    logger.info("Flattening the hierarchical ToC for matching...")
    flat_toc_entries = []
    def _add_ids_and_flatten_recursive(nodes, current_titles_path, counter):
        for node in nodes:
            toc_id = counter[0]
            counter[0] += 1
            title = node.get("title", "").strip()
            if not title: continue
            new_titles_path = current_titles_path + [title]
            entry = {
                "titles_path": new_titles_path,
                "level": node.get("level"),
                "full_title_for_matching": title,
                "toc_id": toc_id
            }
            flat_toc_entries.append(entry)
            if node.get("children"):
                _add_ids_and_flatten_recursive(node.get("children", []), new_titles_path, counter)

    _add_ids_and_flatten_recursive(hierarchical_toc, [], [0])
    logger.info(f"Flattened ToC into {len(flat_toc_entries)} entries.")

    # --- 3. The Core Matching Logic for EPUB (the part you want to see) ---
    logger.info("Assigning metadata to EPUB elements by matching ToC titles...")

    final_documents_with_metadata = []
    toc_titles_for_search = [entry for entry in flat_toc_entries if entry.get("full_title_for_matching")]
    current_hierarchy_metadata = {}

    for element_doc in all_raw_book_docs:
        element_text = element_doc.page_content.strip() if element_doc.page_content else ""
        if not element_text:
            continue

        # Check if this element is a heading that matches a ToC entry
        is_heading = False
        for toc_entry in toc_titles_for_search:
            if element_text == toc_entry["full_title_for_matching"]:
                # It's a heading! Update the current context.
                current_hierarchy_metadata = {"source": os.path.
 ↪basename(BOOK_PATH)}
                for i, title_in_path in enumerate(toc_entry["titles_path"]):
                    current_hierarchy_metadata[f"level_{i+1}_title"] =␣
 ↪title_in_path
                current_hierarchy_metadata['toc_id'] = toc_entry.get('toc_id')
                is_heading = True
                break # Found the match, no need to search further

        # Assign metadata
        if not current_hierarchy_metadata:
            # Content before the first ToC entry (e.g., cover, title page)
            doc_metadata_to_assign = {"source": os.path.basename(BOOK_PATH),␣
 ↪"level_1_title": "EPUB Preamble", "toc_id": -1}
        else:
            doc_metadata_to_assign = current_hierarchy_metadata.copy()

        final_documents_with_metadata.
 ↪append(Document(page_content=element_text, metadata=doc_metadata_to_assign))

    logger.info(f"Processing complete. Generated␣
 ↪{len(final_documents_with_metadata)} documents with assigned metadata.")

    # --- 4. Print the result for inspection ---
    print_header("INSPECTION RESULTS: Documents Before Chunking", char="=")
    print(f"Total documents created: {len(final_documents_with_metadata)}\n")

    for i, doc in enumerate(final_documents_with_metadata[:100]): # Print first␣
 ↪30 to avoid flooding the output
        print(f"--- Document [{i+1}] ---")
        print(f"  Assigned Metadata: {doc.metadata}")
        print(f"  Content (Un-chunked Element):")
        print(f"  >> '{doc.page_content}'")
        print("-" * 25 + "\n")


# --- Execute the inspection ---
inspect_epub_preprocessing()
```

5.1.1 Full Database Health & Hierarchy Diagnostic Report

[ ]:

```
# Cell 5.1: Full Database Health & Hierarchy Diagnostic Report (V5 - with␣
 ↪Content Preview)
```

```python
import os
import json
import logging
import random
from typing import List, Dict, Any

# You might need to install pandas if you haven't already
try:
    import pandas as pd
    pandas_available = True
except ImportError:
    pandas_available = False

try:
    from langchain_chroma import Chroma
    from langchain_ollama.embeddings import OllamaEmbeddings
    from langchain_core.documents import Document
    langchain_available = True
except ImportError:
    langchain_available = False

# Setup Logger
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

# --- HELPER FUNCTIONS ---
def print_header(text: str, char: str = "="):
    """Prints a centered header to the console."""
    print("\n" + char * 80)
    print(text.center(80))
    print(char * 80)

def count_total_chunks(node: Dict) -> int:
    """Recursively counts all chunks in a node and its children."""
    total = node.get('_chunks', 0)
    for child_node in node.get('_children', {}).values():
        total += count_total_chunks(child_node)
    return total

def print_hierarchy_report(node: Dict, indent_level: int = 0):
    """
    Recursively prints the reconstructed hierarchy, sorting by sequential ToC ID.
    """
    sorted_children = sorted(
        node.get('_children', {}).items(),
        key=lambda item: item[1].get('_toc_id', float('inf'))
    )

    for title, child_node in sorted_children:
        prefix = "  " * indent_level + "|-- "
        total_chunks_in_branch = count_total_chunks(child_node)
        direct_chunks = child_node.get('_chunks', 0)
        toc_id = child_node.get('_toc_id', 'N/A')
        print(f"{prefix}{title} [ID: {toc_id}] (Total Chuck in branch: {total_chunks_in_branch}, Direct Chunk: {direct_chunks})")
        print_hierarchy_report(child_node, indent_level + 1)

def find_testable_sections(node: Dict, path: str, testable_list: List):
    """
    Recursively find sections with a decent number of "direct" chunks to test sequence on.
    """
    if node.get('_chunks', 0) > 10 and not node.get('_children'):
        testable_list.append({
            "path": path,
            "toc_id": node.get('_toc_id'),
            "chunk_count": node.get('_chunks')
        })

    for title, child_node in node.get('_children', {}).items():
        new_path = f"{path} -> {title}" if path else title
        find_testable_sections(child_node, new_path, testable_list)


# --- MODIFIED TEST FUNCTION ---
def verify_chunk_sequence_and_content(vector_store: Chroma, hierarchy_tree: Dict):
    """
    Selects a random ToC section, verifies chunk sequence, and displays the reassembled content.
    """
    print_header("Chunk Sequence & Content Integrity Test", char="-")
    logger.info("Verifying chunk order and reassembling content for a random ToC section.")

    # 1. Find a good section to test
    testable_sections = []
    find_testable_sections(hierarchy_tree, "", testable_sections)

    if not testable_sections:
        logger.warning("Could not find a suitable section with enough chunks to test. Skipping content test.")
        return

    random_section = random.choice(testable_sections)
    test_toc_id = random_section['toc_id']
    section_title = random_section['path'].split(' -> ')[-1]

    logger.info(f"Selected random section for testing: '{random_section['path']}' (toc_id: {test_toc_id})")

    # 2. Retrieve all documents (content + metadata) for that toc_id
    try:
        # Use .get() to retrieve full documents, not just similarity search
        retrieved_data = vector_store.get(
            where={"toc_id": test_toc_id},
            include=["metadatas", "documents"]
        )

        # Combine metadatas and documents into LangChain Document objects
        docs = [Document(page_content=doc, metadata=meta) for doc, meta in zip(retrieved_data['documents'], retrieved_data['metadatas'])]

        logger.info(f"Retrieved {len(docs)} document chunks for toc_id {test_toc_id}.")

        if len(docs) < 1:
            logger.warning("No chunks found in the selected section. Skipping.")
            return

        # 3. Sort the documents by chunk_id
        # Handle cases where chunk_id might be missing for robustness
        docs.sort(key=lambda d: d.metadata.get('chunk_id', -1))

        chunk_ids = [d.metadata.get('chunk_id') for d in docs]
        if None in chunk_ids:
            logger.error("TEST FAILED: Some retrieved chunks are missing a 'chunk_id'.")
            return

        # 4. Verify sequence
        is_sequential = all(chunk_ids[i] == chunk_ids[i-1] + 1 for i in range(1, len(chunk_ids)))

        # 5. Reassemble and print content
        full_content = "\n".join([d.page_content for d in docs])

        print("\n" + "-"*25 + " CONTENT PREVIEW " + "-"*25)
        print(f"Title: {section_title} [toc_id: {test_toc_id}]")
        print(f"Chunk IDs: {chunk_ids}")
        print("-" * 70)
        print(full_content)
        print("-" * 23 + " END CONTENT PREVIEW " + "-"*23 + "\n")

        if is_sequential:
            logger.info("  TEST PASSED: Chunk IDs for the section are␣
 ↪sequential and content is reassembled.")
        else:
            logger.warning("TEST PASSED (with note): Chunk IDs are not␣
 ↪perfectly sequential but are in increasing order.")
            logger.warning("This is acceptable. Sorting by chunk_id␣
 ↪successfully restored narrative order.")

    except Exception as e:
        logger.error(f"TEST FAILED: An error occurred during chunk sequence␣
 ↪verification: {e}", exc_info=True)


# --- MAIN DIAGNOSTIC FUNCTION ---
def run_full_diagnostics():
    if not langchain_available:
        logger.error("LangChain components not installed. Skipping diagnostics.
 ↪")
        return
    if not pandas_available:
        logger.warning("Pandas not installed. Some reports may not be available.
 ↪")

    print_header("Full Database Health & Hierarchy Diagnostic Report")

    # 1. Connect to the Database
    logger.info("Connecting to the vector database...")
    if not os.path.exists(CHROMA_PERSIST_DIR):
        logger.error(f"FATAL: Chroma DB directory not found at␣
 ↪{CHROMA_PERSIST_DIR}.")
        return

    vector_store = Chroma(
        persist_directory=CHROMA_PERSIST_DIR,
        embedding_function=OllamaEmbeddings(model=EMBEDDING_MODEL_OLLAMA),
        collection_name=CHROMA_COLLECTION_NAME
    )
    logger.info("Successfully connected to the database.")

    # 2. Retrieve ALL Metadata
    total_docs = vector_store._collection.count()
    if total_docs == 0:
        logger.warning("Database is empty. No diagnostics to run.")
        return

    logger.info(f"Retrieving metadata for all {total_docs} chunks...")
    metadatas = vector_store.get(limit=total_docs,␣
↪include=["metadatas"])['metadatas']
    logger.info("Successfully retrieved all metadata.")

    # 3. Reconstruct the Hierarchy Tree
    logger.info("Reconstructing hierarchy from chunk metadata...")
    hierarchy_tree = {'_children': {}}
    chunks_without_id = 0

    for meta in metadatas:
        toc_id = meta.get('toc_id')
        if toc_id is None or toc_id == -1:
            chunks_without_id += 1
            node_title = meta.get('level_1_title', 'Orphaned Chunks')
            if node_title not in hierarchy_tree['_children']:
                 hierarchy_tree['_children'][node_title] = {'_children': {},␣
↪'_chunks': 0, '_toc_id': float('inf')}
            hierarchy_tree['_children'][node_title]['_chunks'] += 1
            continue

        current_node = hierarchy_tree
        for level in range(1, 7):
            level_key = f'level_{level}_title'
            title = meta.get(level_key)
            if not title: break
            if title not in current_node['_children']:
                current_node['_children'][title] = {'_children': {}, '_chunks':␣
↪0, '_toc_id': float('inf')}
            current_node = current_node['_children'][title]

        current_node['_chunks'] += 1
        current_node['_toc_id'] = min(current_node['_toc_id'], toc_id)

    logger.info("Hierarchy reconstruction complete.")

    # 4. Print Hierarchy Report
    print_header("Reconstructed Hierarchy Report (Book Order)", char="-")
    print_hierarchy_report(hierarchy_tree)

    # 5. Run Chunk Sequence and Content Test
    verify_chunk_sequence_and_content(vector_store, hierarchy_tree)
    # 6. Final Summary
    print_header("Diagnostic Summary", char="-")
    print(f"Total Chunks in DB: {total_docs}")

    if chunks_without_id > 0:
        logger.warning(f"Found {chunks_without_id} chunks MISSING a valid␣
 ↪'toc_id'. Check 'Orphaned' sections.")
    else:
        logger.info("All chunks contain valid 'toc_id' metadata. Sequential␣
 ↪integrity is maintained.")

    print_header("Diagnostic Complete")

# --- Execute Diagnostics ---
if 'CHROMA_PERSIST_DIR' in locals() and langchain_available:
    run_full_diagnostics()
else:
    logger.error("Skipping diagnostics: Global variables not defined or␣
 ↪LangChain not available.")
```

[ ]:

```
# Cell 6: Verify Content Retrieval for a Specific toc_id with Reassembled Text

import os
import json
import logging
from langchain_chroma import Chroma
from langchain_ollama.embeddings import OllamaEmbeddings

# --- Logger Setup ---
logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
 ↪%(message)s')

def retrieve_and_print_chunks_for_toc_id(vector_store: Chroma, toc_id: int):
    """
    Retrieves all chunks for a specific toc_id, reconstructs the section title
    from hierarchical metadata, shows the reassembled text, and lists individual
    chunk details for verification.
    """
    try:
        # Use the 'get' method with a 'where' filter to find all chunks for the␣
 ↪toc_id
        results = vector_store.get(
            where={"toc_id": toc_id},
            include=["documents", "metadatas"]
        )
        if not results or not results.get('ids'):
            logger.warning(f"No chunks found in the database for toc_id =␣
↪{toc_id}")
            print("=" * 80)
            print(f"VERIFICATION FAILED: No content found for toc_id: {toc_id}")
            print("=" * 80)
            return

        documents = results['documents']
        metadatas = results['metadatas']

        # --- FIX START: Reconstruct the hierarchical section title from␣
↪metadata ---
        # We assume all chunks for the same toc_id share the same titles.
        # We will inspect the metadata of the first chunk to get the title.
        section_title = "Unknown or Uncategorized Section"
        if metadatas:
            first_meta = metadatas[0]

            # Find all 'level_X_title' keys in the metadata
            level_titles = []
            for key, value in first_meta.items():
                if key.startswith("level_") and key.endswith("_title"):
                    try:
                        # Extract the level number (e.g., 1 from␣
↪'level_1_title') for sorting
                        level_num = int(key.split('_')[1])
                        level_titles.append((level_num, value))
                    except (ValueError, IndexError):
                        # Ignore malformed keys, just in case
                        continue

            # Sort the titles by their level number (1, 2, 3...)
            level_titles.sort()

            # Join the sorted titles to create a breadcrumb-style title
            if level_titles:
                title_parts = [title for num, title in level_titles]
                section_title = " > ".join(title_parts)
        # --- FIX END ---

        # --- Print a clear header with the reconstructed section title ---
        print("=" * 80)
        print(f"VERIFYING SECTION: '{section_title}' (toc_id: {toc_id})")
        print("=" * 80)
        logger.info(f"Found {len(documents)} chunks in the database for this section.")

        # Sort chunks by their chunk_id to ensure they are in the correct order for reassembly
        sorted_items = sorted(zip(documents, metadatas), key=lambda item: item[1].get('chunk_id', 0))

        # --- Reassemble and print the full text for the section ---
        all_chunk_texts = [item[0] for item in sorted_items]
        reassembled_text = "\n".join(all_chunk_texts)

        print("\n" + "#" * 28 + " Reassembled Text " + "#" * 28)
        print(reassembled_text)
        print("#" * 80)

        # --- Print individual chunk details for in-depth verification ---
        print("\n" + "-" * 24 + " Retrieved Chunk Details " + "-" * 25)
        for i, (doc, meta) in enumerate(sorted_items):
            print(f"\n[ Chunk {i+1} of {len(documents)} | chunk_id: {meta.get('chunk_id', 'N/A')} ]")
            content_preview = doc.replace('\n', ' ').strip()
            print(f"  Content Preview: '{content_preview[:250]}...'")
            print(f"  Metadata: {json.dumps(meta, indent=2)}")

        print("\n" + "=" * 80)
        print(f"Verification complete for section '{section_title}'.")
        print("=" * 80)

    except Exception as e:
        logger.error(f"An error occurred during retrieval for toc_id {toc_id}: {e}", exc_info=True)

# ==============================================================================
# EXECUTION BLOCK (No changes needed here)
# ==============================================================================

# --- IMPORTANT: Set the ID of the section you want to test here ---
# Example: ToC ID 10 might be "An Overview of Digital Forensics"
# Example: ToC ID 11 might be "Digital Forensics and Other Related Disciplines"
TOC_ID_TO_TEST = 9# Change this to an ID you know exists from your ToC


# Assume these variables are defined in a previous cell from your notebook
# CHROMA_PERSIST_DIR = "./chroma_db_with_metadata"
# EMBEDDING_MODEL_OLLAMA = "nomic-embed-text"
# CHROMA_COLLECTION_NAME = "forensics_handbook"
```

```python
# Check if the database directory exists before attempting to connect
if 'CHROMA_PERSIST_DIR' in locals() and os.path.exists(CHROMA_PERSIST_DIR):
    logger.info(f"Connecting to the existing vector database at␣
 ↪'{CHROMA_PERSIST_DIR}'...")

    try:
        vector_store = Chroma(
            persist_directory=CHROMA_PERSIST_DIR,
            embedding_function=OllamaEmbeddings(model=EMBEDDING_MODEL_OLLAMA),
            collection_name=CHROMA_COLLECTION_NAME
        )

        # Run the verification function
        retrieve_and_print_chunks_for_toc_id(vector_store, TOC_ID_TO_TEST)

    except Exception as e:
        logger.error(f"Failed to initialize Chroma or run retrieval. Error:␣
 ↪{e}")
        logger.error("Please ensure your embedding model and collection names␣
 ↪are correct.")

else:
    logger.error("Database directory not found or 'CHROMA_PERSIST_DIR' variable␣
 ↪is not set.")
    logger.error("Please run the previous cell (Cell 5) to create the database␣
 ↪first.")
```

5.2 Test Data Base for content development

Require Description

[ ]:

```python
# Cell 7: Verify Vector Database (Final Version with Rich Diagnostic Output)

import os
import json
import re
import random
import logging
from typing import List, Dict, Any, Tuple, Optional

# Third-party imports
try:
    from langchain_chroma import Chroma
    from langchain_ollama.embeddings import OllamaEmbeddings
    from langchain_core.documents import Document
    langchain_available = True
```

```
except ImportError:
    langchain_available = False

# Setup Logger for this cell
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
 ↪%(message)s')
logger = logging.getLogger(__name__)

# --- HELPER FUNCTIONS ---


def print_results(query_text: str, results: list, where_filter: Optional[Dict]␣
 ↪= None):
    """
    Richly prints query results, showing the query, filter, and retrieved␣
 ↪documents.
    """
    print("\n" + "-"*10 + " DIAGNOSTIC: RETRIEVAL RESULTS " + "-"*10)
    print(f"QUERY: '{query_text}'")
    if where_filter:
        print(f"FILTER: {json.dumps(where_filter, indent=2)}")

    if not results:
        print("--> No documents were retrieved for this query and filter.")
        print("-" * 55)
        return

    print(f"--> Found {len(results)} results. Displaying top {min(len(results),␣
 ↪3)}:")
    for i, doc in enumerate(results[:3]):
        print(f"\n[ RESULT {i+1} ]")
        content_preview = doc.page_content.replace('\n', ' ').strip()
        print(f"  Content : '{content_preview[:200]}...'")
        print(f"  Metadata: {json.dumps(doc.metadata, indent=2)}")
    print("-" * 55)


# --- HELPER FUNCTIONS FOR FINDING DATA (UNCHANGED) ---
def find_deep_entry(nodes: List[Dict], current_path: List[str] = []) ->␣
 ↪Optional[Tuple[Dict, List[str]]]:
    shuffled_nodes = random.sample(nodes, len(nodes))
    for node in shuffled_nodes:
        if node.get('level', 0) >= 2 and node.get('children'): return node,␣
 ↪current_path + [node['title']]
        if node.get('children'):
            path = current_path + [node['title']]
            deep_entry = find_deep_entry(node['children'], path)
            if deep_entry: return deep_entry
    return None

def find_chapter_title_by_number(toc_data: List[Dict], chap_num: int) ->␣
 ↪Optional[List[str]]:
    def search_nodes(nodes, num, current_path):
        for node in nodes:
            path = current_path + [node['title']]
            if re.match(rf"(Chapter\s)?{num}[.:\s]", node.get('title', ''), re.
 ↪IGNORECASE): return path
            if node.get('children'):
                found_path = search_nodes(node['children'], num, path)
                if found_path: return found_path
        return None
    return search_nodes(toc_data, chap_num, [])


# --- ENHANCED TEST CASES with DIAGNOSTIC OUTPUT ---

def basic_retrieval_test(db, outline):
    print_header("Test 1: Basic Retrieval", char="-")
    try:
        logger.info("Goal: Confirm the database is live and contains␣
 ↪thematically relevant content.")
        logger.info("Strategy: Perform a simple similarity search using the␣
 ↪course's 'unitName'.")
        query_text = outline.get("unitInformation", {}).get("unitName",␣
 ↪"introduction")

        logger.info(f"Action: Searching for query: '{query_text}'...")
        results = db.similarity_search(query_text, k=1)

        print_results(query_text, results) # <--- SHOW THE EVIDENCE

        logger.info("Verification: Check if at least one document was returned.
 ↪")
        assert len(results) > 0, "Basic retrieval query returned no results."

        logger.info("  Result: TEST 1 PASSED. The database is online and␣
 ↪responsive.")
        return True
    except Exception as e:
        logger.error(f"  Result: TEST 1 FAILED. Reason: {e}")
        return False
```

```
def deep_hierarchy_test(db, toc):
    print_header("Test 2: Deep Hierarchy Retrieval", char="-")
    try:
        logger.info("Goal: Verify that the multi-level hierarchical metadata␣
 ↪was ingested correctly.")
        logger.info("Strategy: Find a random, deeply nested sub-section and use␣
 ↪a precise filter to retrieve it.")
        deep_entry_result = find_deep_entry(toc)
        assert deep_entry_result, "Could not find a suitable deep entry (level␣
 ↪>= 2) to test."
        node, path = deep_entry_result
        query = node['title']

        logger.info(f"  - Selected random deep section: {' -> '.join(path)}")
        conditions = [{f"level_{i+1}_title": {"$eq": title}} for i, title in␣
 ↪enumerate(path)]
        w_filter = {"$and": conditions}

        logger.info("Action: Performing a similarity search with a highly␣
 ↪specific '$and' filter.")
        results = db.similarity_search(query, k=1, filter=w_filter)

        print_results(query, results, w_filter) # <--- SHOW THE EVIDENCE

        logger.info("Verification: Check if the precisely filtered query␣
 ↪returned any documents.")
        assert len(results) > 0, "Deeply filtered query returned no results."

        logger.info("  Result: TEST 2 PASSED. Hierarchical metadata is␣
 ↪structured correctly.")
        return True
    except Exception as e:
        logger.error(f"  Result: TEST 2 FAILED. Reason: {e}")
        return False

def advanced_alignment_test(db, outline, toc):
    print_header("Test 3: Advanced Unit Outline Alignment", char="-")
    try:
        logger.info("Goal: Ensure a weekly topic from the syllabus can be␣
 ↪mapped to the correct textbook chapter(s).")
        logger.info("Strategy: Pick a random week, find its chapter, and query␣
 ↪for the topic filtered by that chapter.")
        week_to_test = random.choice(outline['weeklySchedule'])
        logger.info(f"  - Selected random week: Week {week_to_test['week']} -␣
 ↪'{week_to_test['contentTopic']}'")
```

```
        reading = week_to_test.get('requiredReading', '')
        chap_nums_str = re.findall(r'\d+', reading)
        assert chap_nums_str, f"Could not find chapter numbers in required␣
 ↪reading: '{reading}'"
        logger.info(f"  - Extracted required chapter number(s):␣
 ↪{chap_nums_str}")

        chapter_paths = [find_chapter_title_by_number(toc, int(n)) for n in␣
 ↪chap_nums_str]
        chapter_paths = [path for path in chapter_paths if path is not None]
        assert chapter_paths, f"Could not map chapter numbers {chap_nums_str}␣
 ↪to a valid ToC path."

        level_1_titles = list(set([path[0] for path in chapter_paths]))
        logger.info(f"  - Mapped to top-level ToC entries: {level_1_titles}")

        or_filter = [{"level_1_title": {"$eq": title}} for title in␣
 ↪level_1_titles]
        w_filter = {"$or": or_filter} if len(or_filter) > 1 else or_filter[0]
        query = week_to_test['contentTopic']

        logger.info("Action: Searching for the weekly topic, filtered by the␣
 ↪mapped chapter(s).")
        results = db.similarity_search(query, k=5, filter=w_filter)

        print_results(query, results, w_filter) # <--- SHOW THE EVIDENCE

        logger.info("Verification: Check if at least one returned document is␣
 ↪from the correct chapter.")
        assert len(results) > 0, "Alignment query returned no results for the␣
 ↪correct section/chapter."

        logger.info("  Result: TEST 3 PASSED. The syllabus can be reliably␣
 ↪aligned with the textbook content.")
        return True
    except Exception as e:
        logger.error(f"  Result: TEST 3 FAILED. Reason: {e}")
        return False

def content_sequence_test(db, outline):
    print_header("Test 4: Content Sequence Verification", char="-")
    try:
        logger.info("Goal: Confirm that chunks for a topic can be re-ordered to␣
 ↪form a coherent narrative.")
        logger.info("Strategy: Retrieve several chunks for a random topic and␣
 ↪verify their 'chunk_id' is sequential.")
```

```
        topic_query = random.choice(outline['weeklySchedule'])['contentTopic']

        logger.info(f"Action: Performing similarity search for topic:␣
 ↪'{topic_query}' to get a set of chunks.")
        results = db.similarity_search(topic_query, k=10)

        print_results(topic_query, results) # <--- SHOW THE EVIDENCE

        docs_with_id = [doc for doc in results if 'chunk_id' in doc.metadata]
        assert len(docs_with_id) > 3, "Fewer than 4 retrieved chunks have a␣
 ↪'chunk_id' to test."

        chunk_ids = [doc.metadata['chunk_id'] for doc in docs_with_id]
        sorted_ids = sorted(chunk_ids)

        logger.info(f"  - Retrieved and sorted chunk IDs: {sorted_ids}")
        logger.info("Verification: Check if the sorted list of chunk_ids is␣
 ↪strictly increasing.")
        is_ordered = all(sorted_ids[i] >= sorted_ids[i-1] for i in range(1,␣
 ↪len(sorted_ids)))
        assert is_ordered, "The retrieved chunks' chunk_ids are not in␣
 ↪ascending order when sorted."

        logger.info("  Result: TEST 4 PASSED. Narrative order can be␣
 ↪reconstructed using 'chunk_id'.")
        return True
    except Exception as e:
        logger.error(f"  Result: TEST 4 FAILED. Reason: {e}")
        return False

# --- MAIN VERIFICATION EXECUTION ---
def run_verification():
    print_header("Database Verification Process")

    if not langchain_available:
        logger.error("LangChain libraries not found. Aborting tests.")
        return

    required_files = {
        "Chroma DB": CHROMA_PERSIST_DIR,
        "ToC JSON": PRE_EXTRACTED_TOC_JSON_PATH,
        "Parsed Outline": PARSED_UO_JSON_PATH
    }
    for name, path in required_files.items():
        if not os.path.exists(path):
            logger.error(f"Required '{name}' not found at '{path}'. Please run previous cells.")
            return

    with open(PRE_EXTRACTED_TOC_JSON_PATH, 'r', encoding='utf-8') as f:
        toc_data = json.load(f)
    with open(PARSED_UO_JSON_PATH, 'r', encoding='utf-8') as f:
        unit_outline_data = json.load(f)

    logger.info("Connecting to DB and initializing components...")
    embeddings = OllamaEmbeddings(model=EMBEDDING_MODEL_OLLAMA)
    vector_store = Chroma(
        persist_directory=CHROMA_PERSIST_DIR,
        embedding_function=embeddings,
        collection_name=CHROMA_COLLECTION_NAME
    )

    results_summary = [
        basic_retrieval_test(vector_store, unit_outline_data),
        deep_hierarchy_test(vector_store, toc_data),
        advanced_alignment_test(vector_store, unit_outline_data, toc_data),
        content_sequence_test(vector_store, unit_outline_data)
    ]

    passed_count = sum(filter(None, results_summary))
    failed_count = len(results_summary) - passed_count

    print_header("Verification Summary")
    print(f"Total Tests Run: {len(results_summary)}")
    print(f"  Passed: {passed_count}")
    print(f"  Failed: {failed_count}")
    print_header("Verification Complete", char="=")

# --- Execute Verification ---
# Assumes global variables from Cell 1 are available in the notebook's scope
run_verification()
```

6 Content Generation

6.1 Planning Agent

[ ]:

```
# Cell 8: The Data-Driven Planning Agent (Final Hierarchical Version )

import os
import json
import re
import math
import logging
from typing import List, Dict, Any, Optional, Tuple

# Setup Logger and LangChain components
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
 ↪%(message)s')
logger = logging.getLogger(__name__)
try:
    from langchain_chroma import Chroma
    from langchain_ollama.embeddings import OllamaEmbeddings
    langchain_available = True
except ImportError:
    langchain_available = False

def print_header(text: str, char: str = "="):
    """Prints a centered header to the console."""
    print("\n" + char * 80)
    print(text.center(80))
    print(char * 80)

class PlanningAgent:
    """
    An agent that creates a hierarchical content plan, adaptively partitions␣
 ↪content
    into distinct lecture decks, and allocates presentation time.
    """
    def __init__(self, master_config: Dict, vector_store: Optional[Any] = None):
        self.config = master_config['processed_settings']
        self.unit_outline = master_config['unit_outline']
        self.book_toc = master_config['book_toc']
        self.flat_toc_with_ids = self._create_flat_toc_with_ids()
        self.vector_store = vector_store
        logger.info("Data-Driven PlanningAgent initialized successfully.")

    def _create_flat_toc_with_ids(self) -> List[Dict]:
        """Creates a flattened list of the ToC for easy metadata lookup."""
        flat_list = []
        def flatten_recursive(nodes, counter):
            for node in nodes:
                node_id = counter[0]; counter[0] += 1
                flat_list.append({'toc_id': node_id, 'title': node.get('title',␣
 ↪''), 'node': node})
                if node.get('children'):
                    flatten_recursive(node.get('children'), counter)
        flatten_recursive(self.book_toc, [0])
        return flat_list
    def _assign_sequence_ids_recursively(self, node: Dict, counter: list):
        """
        Recursively walks a content node, assigning a unique, sequential ID
        to the node, its children, and its interactive activity in a depth-first order.
        """
        # 1. Assign sequence ID to the parent content node itself.
        node['seq_id'] = counter[0]
        counter[0] += 1

        # 2. Recurse through all children first.
        if 'children' in node and node['children']:
            for child in node['children']:
                self._assign_sequence_ids_recursively(child, counter)

        # 3. Assign an ID to the interactive activity LAST, so it appears after the content and its children.
        if 'interactive_activity' in node:
            node['interactive_activity']['seq_id'] = counter[0]
            counter[0] += 1

    def _identify_relevant_chapters(self, weekly_schedule_item: Dict) -> List[int]:
        """Extracts chapter numbers precisely from the 'requiredReading' string."""
        reading_str = weekly_schedule_item.get('requiredReading', '')
        match = re.search(r'Chapter(s)?', reading_str, re.IGNORECASE)
        if not match: return []
        search_area = reading_str[match.start():]
        chap_nums_str = re.findall(r'\d+', search_area)
        if chap_nums_str:
            return sorted(list(set(int(n) for n in chap_nums_str)))
        return []

    def _find_chapter_node(self, chapter_number: int) -> Optional[Dict]:
        """Finds the ToC node for a specific chapter number."""
        for item in self.flat_toc_with_ids:
            if re.match(rf"Chapter\s{chapter_number}(?:\D|$)", item['title']):
                return item['node']
        return None

    def _build_topic_plan_tree(self, toc_node: Dict) -> Dict:
        """
        Recursively builds a hierarchical plan tree from any ToC node,
        annotating it with direct and total branch chunk counts.
        """
```

```python
        node_metadata = next((item for item in self.flat_toc_with_ids if item['node'] is toc_node), None)
        if not node_metadata: return {}

        retrieved_docs = self.vector_store.get(where={'toc_id': node_metadata['toc_id']})
        direct_chunk_count = len(retrieved_docs.get('ids', []))

        plan_node = {
            "title": node_metadata['title'],
            "toc_id": node_metadata['toc_id'],
            "chunk_count": direct_chunk_count,
            "total_chunks_in_branch": 0,
            "slides_allocated": 0,
            "children": []
        }

        child_branch_total = 0
        for child_node in toc_node.get('children', []):
            if any(ex in child_node.get('title', '').lower() for ex in ["review", "introduction", "summary", "key terms"]):
                continue
            child_plan_node = self._build_topic_plan_tree(child_node)
            if child_plan_node:
                plan_node['children'].append(child_plan_node)
                child_branch_total += child_plan_node.get('total_chunks_in_branch', 0)

        plan_node['total_chunks_in_branch'] = direct_chunk_count + child_branch_total
        return plan_node

    # In PlanningAgent Class...

    def _allocate_slides_to_tree(self, plan_tree: Dict, content_slides_budget: int):
        """
        (FINAL, REORDERED FOR CLARITY) Performs a multi-pass process to allocate content slides,
        add activities, sum totals, and reorders the keys in each node for maximum readability.
        """
        if not plan_tree or content_slides_budget <= 0:
            return plan_tree

        # --- Pass 1: Allocate Content Slides ---
        def allocate_content_recursively(node, budget):
            node['budget_slides_content'] = round(budget)
            node['direct_slides_content'] = 0
            if not node.get('children'):
                node['direct_slides_content'] = round(budget)
                return
            total_branch_chunks = node.get('total_chunks_in_branch', 0)
            own_content_slides = 0
            if total_branch_chunks > 0:
                own_content_slides = round(budget * (node.get('chunk_count', 0) / total_branch_chunks))
            node['direct_slides_content'] = own_content_slides
            remaining_budget_for_children = budget - own_content_slides
            children_total_chunks = total_branch_chunks - node.get('chunk_count', 0)
            if children_total_chunks <= 0: return
            for child in node.get('children', []):
                child_budget = remaining_budget_for_children * (child.get('total_chunks_in_branch', 0) / children_total_chunks)
                allocate_content_recursively(child, child_budget)

        allocate_content_recursively(plan_tree, content_slides_budget)

        # --- Pass 2: Add Interactive Activities ---
        def add_interactive_nodes(node, depth, interactive_deep):
            if not node: return
            if self.config.get('interactive', False):
                if interactive_deep:
                    if depth == 2: node['interactive_activity'] = {"title": f"{node.get('title')} (Deep-Dive Activity)", "toc_id": node.get('toc_id'), "slides_allocated": 1}
                    if depth == 1: node['interactive_activity'] = {"title": f"{node.get('title')} (General Activity)", "toc_id": node.get('toc_id'), "slides_allocated": 1}
                else:
                    if depth == 1: node['interactive_activity'] = {"title": f"{node.get('title')} (Interactive Activity)", "toc_id": node.get('toc_id'), "slides_allocated": 1}
            for child in node.get('children', []):
                add_interactive_nodes(child, depth + 1, interactive_deep)

        add_interactive_nodes(plan_tree, 1, self.config.get('interactive_deep', False))

        # --- Pass 3: Sum All Slides Up the Tree ---
        def sum_slides_upwards(node):
            total_slides = node.get('direct_slides_content', 0)
            total_slides += node.get('interactive_activity', {}).
↪get('slides_allocated', 0)
            if node.get('children'):
                total_slides += sum(sum_slides_upwards(child) for child in node.
↪get('children', []))
            node['total_slides_in_branch'] = total_slides
            return total_slides

        sum_slides_upwards(plan_tree)

        # --- NEW: Pass 4: Reorder keys for final clarity ---
        def reorder_keys_for_readability(node: Dict) -> Dict:
            if not node:
                return None

            # Define the desired order of keys
            key_order = [
                "title",
                "toc_id",
                "chunk_count",
                "total_chunks_in_branch",
                "budget_slides_content",
                "direct_slides_content",
                "total_slides_in_branch",
                "children",
                "interactive_activity"

            ]

            # Rebuild the dictionary in the specified order
            reordered_node = {key: node[key] for key in key_order if key in␣
↪node}

            # Recursively reorder children
            if 'children' in reordered_node:
                reordered_node['children'] =␣
↪[reorder_keys_for_readability(child) for child in reordered_node['children']]

            return reordered_node

        return reorder_keys_for_readability(plan_tree)

    def create_content_plan_for_week(self, week_number: int) -> Optional[Dict]:
        """Orchestrates the adaptive planning and partitioning process."""
        print_header(f"Planning Week {week_number}", char="*")
        weekly_schedule_item = self.unit_outline['weeklySchedule'][week_number␣
↪- 1]
        chapter_numbers = self._identify_relevant_chapters(weekly_schedule_item)
        if not chapter_numbers: return None

        num_decks = self.config['week_session_setup'].get('sessions_per_week',␣
↪1)

        # 1. Build a full plan tree for each chapter to get its weight.
        chapter_plan_trees = [self._build_topic_plan_tree(self.
↪_find_chapter_node(cn)) for cn in chapter_numbers if self.
↪_find_chapter_node(cn)]
        total_weekly_chunks = sum(tree.get('total_chunks_in_branch', 0) for␣
↪tree in chapter_plan_trees)

        # 2. NEW: Adaptive Partitioning Strategy
        partitionable_units = []
        all_top_level_sections = []
        for chapter_tree in chapter_plan_trees:
            all_top_level_sections.extend(chapter_tree.get('children', []))

        num_top_level_sections = len(all_top_level_sections)

        # Always prefer to split by top-level sections if there are enough to␣
↪distribute.
        if num_top_level_sections >= num_decks:
            logger.info(f"Partitioning strategy: Distributing␣
↪{num_top_level_sections} top-level sections across {num_decks} decks.")
            partitionable_units = all_top_level_sections
        else:
            # Fallback for rare cases where there are fewer topics than decks␣
↪(e.g., 1 chapter with 1 section, but 2 decks).
            logger.info(f"Partitioning strategy: Not enough top-level sections␣
↪({num_top_level_sections}) to fill all decks ({num_decks}). Distributing␣
↪whole chapters instead.")
            partitionable_units = chapter_plan_trees

        # 3. Partition the chosen units into decks using a bin-packing algorithm
        decks = [[] for _ in range(num_decks)]
        deck_weights = [0] * num_decks
        sorted_units = sorted(partitionable_units, key=lambda x: x.
↪get('toc_id', 0))

        for unit in sorted_units:
            lightest_deck_index = deck_weights.index(min(deck_weights))
            decks[lightest_deck_index].append(unit)
            deck_weights[lightest_deck_index] += unit.
↪get('total_chunks_in_branch', 0)

        # 4. Plan each deck
        content_slides_per_week = self.config['slide_count_strategy'].
↪get('target_total_slides', 25)
        final_deck_plans = []
        for i, deck_content_trees in enumerate(decks):
            deck_number = i + 1
            deck_chunk_weight = sum(tree.get('total_chunks_in_branch', 0) for␣
↪tree in deck_content_trees)
            deck_slide_budget = round((deck_chunk_weight / total_weekly_chunks)␣
↪* content_slides_per_week) if total_weekly_chunks > 0 else 0

            logger.info(f"--- Planning Deck {deck_number}/{num_decks} | Topics:␣
↪{[t['title'] for t in deck_content_trees]} | Weight: {deck_chunk_weight}␣
↪chunks | Slide Budget: {deck_slide_budget} ---")

            # The allocation function is recursive and works on any tree or␣
↪sub-tree
            planned_content = [self._allocate_slides_to_tree(tree,␣
↪round(deck_slide_budget * (tree.get('total_chunks_in_branch', 0) /␣
↪deck_chunk_weight))) if deck_chunk_weight > 0 else tree for tree in␣
↪deck_content_trees]

            final_deck_plans.append({
                "deck_number": deck_number,
                "deck_title": f"{self.config.get('unit_name', 'Course')} - Week␣
↪{week_number}, Lecture {deck_number}",
                "session_content": planned_content
            })

        return {
            "week": week_number,
            "overall_topic": weekly_schedule_item.get('contentTopic'),
            "deck_plans": final_deck_plans
        }


    def finalize_and_calculate_time_plan(self,draft_plan: Dict, config: Dict)␣
↪-> Dict:
        """
        Takes a draft plan and enriches it by:
        1. Calculating detailed slide counts and time allocations for every␣
↪node.
        2. Adding framework sections and wrapping content.
        3. Calculating and adding summaries for decks and the entire week.
        4. Assigning a final sequence ID (seq_id) for slide generation.
        5. Reordering all keys for maximum readability.
        """
        final_plan = json.loads(json.dumps(draft_plan))

        # --- Time Constants from Config ---
        params = config.get('parameters_slides', {})
        TIME_PER_CONTENT = params.get('time_per_content_slides_min', 3)
        TIME_PER_INTERACTIVE = params.get('time_per_interactive_slide_min', 5)
        TIME_FOR_FRAMEWORK_DECK = params.get('time_for_framework_slides_min', 6)
        FRAMEWORK_SLIDES_PER_DECK = 4


        # --- Recursive Helper Functions ---
        def _calculate_time_and_reorder(node: Dict):
            # 1. Recurse to the bottom first to perform a bottom-up calculation
            children_total_time = 0
            if 'children' in node and node['children']:
                for child in node['children']:
                    _calculate_time_and_reorder(child) # Recursive call
                    children_total_time += child.get('time_allocation_minutes', {}).get('total_branch_time', 0)

            # 2. Calculate this node's direct time
            direct_content_time = node.get('direct_slides_content', 0) * TIME_PER_CONTENT
            interactive_time = node.get('interactive_activity', {}).get('slides_allocated', 0) * TIME_PER_INTERACTIVE

            # 3. Calculate this node's total branch time
            branch_total_time = direct_content_time + interactive_time + children_total_time

            # 4. Create the time allocation object
            time_alloc = {
                "direct_content_time": direct_content_time,
                "direct_interactive_time": interactive_time,
                "total_branch_time": branch_total_time
            }
            node['time_allocation_minutes'] = time_alloc

            # 5. Reorder all keys for this node to ensure final clarity
            key_order = [
                "title",
                "toc_id",
                "chunk_count",
```

```python
                "total_chunks_in_branch",
                "budget_slides_content",
                "direct_slides_content",
                "total_slides_in_branch",
                "time_allocation_minutes",
                "children",
                "interactive_activity"

            ]
            reordered_node = {key: node[key] for key in key_order if key in node}

            # Clear the original node and update it with the reordered keys
            node.clear()
            node.update(reordered_node)

        # --- Main Processing Loop for Decks ---
        for deck in final_plan.get("deck_plans", []):
            session_content_blocks = deck.pop("session_content", [])

            # Perform the combined time calculation and reordering pass
            for block in session_content_blocks:
                _calculate_time_and_reorder(block)

            # Create Framework Sections
            week_number, deck_number = final_plan.get("week"), deck.get("deck_number")
            title_section = {"section_type": "Title", "content": { "unit_name": config.get('unit_name', 'Course'), "unit_code": config.get('course_id', ''), "week_topic": final_plan.get('overall_topic', ''), "deck_title": f"Week {week_number}, Lecture {deck_number}"}}
            agenda_section = {"section_type": "Agenda", "content": {"title": "Today's Agenda", "items": [item.get('title', 'Untitled Topic') for item in session_content_blocks]}}
            summary_section = {"section_type": "Summary", "content": {"title": "Summary & Key Takeaways", "placeholder": "Auto-generate based on covered topics."}}
            end_section = {"section_type": "End", "content": {"title": "Thank You", "text": "Questions?"}}
            main_content_block = {"section_type": "Content", "content_blocks": session_content_blocks}

            final_sections_for_deck = [title_section, agenda_section, main_content_block, summary_section, end_section]
            # ****************************************************************
            # *** NEW: Assign Sequential IDs for the entire deck ***
            # ****************************************************************
            seq_counter = [0] # Use a list for pass-by-reference behavior
            for section in final_sections_for_deck:
                if section.get("section_type") == "Content":
                    # For the main content, iterate through its blocks and␣
↪start the recursion
                    for block in section.get("content_blocks", []):
                        self._assign_sequence_ids_recursively(block,␣
↪seq_counter)
                else:
                    # For simple framework slides, just assign the next ID
                    section['seq_id'] = seq_counter[0]
                    seq_counter[0] += 1
            # ****************************************************************
            # *** END NEW CODE ***
            # ****************************************************************


            # Calculate Deck Summaries
            total_content_slides = sum(b.get('total_slides_in_branch', 0) - b.
↪get('interactive_activity',{}).get('slides_allocated',0) for b in␣
↪session_content_blocks)
            total_interactive_slides = sum(b.get('interactive_activity',{}).
↪get('slides_allocated',0) for b in session_content_blocks)

            deck_content_time = sum(b.get('time_allocation_minutes', {}).
↪get('total_branch_time', 0) for b in session_content_blocks)

            deck['total_slides_in_deck'] = FRAMEWORK_SLIDES_PER_DECK + sum(b.
↪get('total_slides_in_branch', 0) for b in session_content_blocks)
            deck['slide_count_breakdown'] = {"framework":␣
↪FRAMEWORK_SLIDES_PER_DECK, "content": total_content_slides, "interactive":␣
↪total_interactive_slides}
            deck['time_breakdown_minutes'] = {"framework":␣
↪TIME_FOR_FRAMEWORK_DECK, "content_and_interactive": deck_content_time,␣
↪"total_deck_time": TIME_FOR_FRAMEWORK_DECK + deck_content_time}
            deck['sections'] = final_sections_for_deck
            if 'deck_title' in deck: del deck['deck_title']

        # --- Calculate Grand Totals for the Week ---
        weekly_slide_summary = {"total_slides_for_week": 0,␣
↪"total_framework_slides": 0, "total_content_slides": 0,␣
↪"total_interactive_slides": 0, "number_of_decks": len(final_plan.
↪get("deck_plans", []))}
```

```
        weekly_time_summary = {"total_time_for_week_minutes": 0, "total_framework_time": 0, "total_content_and_interactive_time": 0}

        for deck in final_plan.get("deck_plans", []):
            weekly_slide_summary['total_slides_for_week'] += deck.get('total_slides_in_deck', 0)
            for key, value in deck.get('slide_count_breakdown', {}).items(): weekly_slide_summary[f"total_{key}_slides"] += value
            weekly_time_summary['total_time_for_week_minutes'] += deck.get('time_breakdown_minutes', {}).get('total_deck_time', 0)
            weekly_time_summary['total_framework_time'] += deck.get('time_breakdown_minutes', {}).get('framework', 0)
            weekly_time_summary['total_content_and_interactive_time'] += deck.get('time_breakdown_minutes', {}).get('content_and_interactive', 0)

        # --- Construct Final Ordered Plan ---
        final_ordered_plan = {
            "week": final_plan.get("week"),
            "overall_topic": final_plan.get("overall_topic"),
            "weekly_slide_summary": weekly_slide_summary,
            "weekly_time_summary_minutes": weekly_time_summary,
            "deck_plans": final_plan.get("deck_plans", [])
        }

        return final_ordered_plan

    # --- NEW FUNCTION TO GENERATE MASTER SUMMARY ---
    def generate_and_save_master_plan(self, weekly_plans: List[Dict], config: Dict):
        """
        Aggregates summaries from all weekly plans into a single master plan file,
        including new grand total metrics.
        """
        print_header("Phase 4: Generating Master Unit Plan", char="#")

        # Initialize the master plan structure with the new fields
        master_plan = {
            "unit_code": config.get('course_id', 'UNKNOWN'),
            "unit_name": config.get('unit_name', 'Unknown Unit'),
            "grand_total_summary": {
                "total_slides_for_unit": 0,
                "total_framework_slides": 0,
                "total_content_slides": 0,
                "total_interactive_slides": 0,
                "total_number_of_decks": 0,
                "total_time_for_unit_minutes": 0,
                "total_time_for_unit_in_hour": 0, # New
                "average_deck_time_in_min": 0,
                "average_deck_time_in_hour": 0
            },
            "weekly_summaries": []
        }

        grand_totals = master_plan["grand_total_summary"]

        # Loop through each weekly plan to aggregate data
        for plan in sorted(weekly_plans, key=lambda p: p.get('week', 0)):
            # Extract the high-level summary for this week
            summary_entry = {
                "week": plan.get("week"),
                "overall_topic": plan.get("overall_topic"),
                "slide_summary": plan.get("weekly_slide_summary"),
                "time_summary_minutes": plan.get("weekly_time_summary_minutes")
            }
            master_plan["weekly_summaries"].append(summary_entry)

            # Add this week's totals to the grand totals
            slide_summary = plan.get("weekly_slide_summary", {})
            time_summary = plan.get("weekly_time_summary_minutes", {})

            grand_totals["total_slides_for_unit"] += slide_summary.
↪get("total_slides_for_week", 0)
            grand_totals["total_framework_slides"] += slide_summary.
↪get("total_framework_slides", 0)
            grand_totals["total_content_slides"] += slide_summary.
↪get("total_content_slides", 0)
            grand_totals["total_interactive_slides"] += slide_summary.
↪get("total_interactive_slides", 0)
            grand_totals["total_number_of_decks"] += slide_summary.
↪get("number_of_decks", 0)
            grand_totals["total_time_for_unit_minutes"] += time_summary.
↪get("total_time_for_week_minutes", 0)

        # --- NEW: Calculate the final derived grand totals after the loop ---
        if grand_totals["total_time_for_unit_minutes"] > 0:
            grand_totals["total_time_for_unit_in_hour"] =␣
↪round(grand_totals["total_time_for_unit_minutes"] / 60, 2)

        if grand_totals["total_number_of_decks"] > 0:
            grand_totals["average_deck_time_in_min"] =␣
 ↪round(grand_totals["total_time_for_unit_minutes"] /␣
 ↪grand_totals["total_number_of_decks"], 2)

        if grand_totals["total_number_of_decks"] > 0:
            grand_totals["average_deck_time_in_hour"] =␣
 ↪round((grand_totals["total_time_for_unit_minutes"] /␣
 ↪grand_totals["total_number_of_decks"]) / 60, 2)


        master_filename = f"{config.get('course_id', 'UNIT')}_master_plan_unit.
 ↪json"
        output_path = os.path.join(PLAN_OUTPUT_DIR, master_filename)

        try:
            with open(output_path, 'w') as f:
                json.dump(master_plan, f, indent=2)
            logger.info(f"Successfully generated and saved Master Unit Plan to:␣
 ↪{output_path}")
            print("\n--- Preview of Master Plan ---")
            print(json.dumps(master_plan, indent=2))
            return True
        except Exception as e:
            logger.error(f"Failed to save Master Unit Plan: {e}", exc_info=True)
```

6.2 Content Generator Agent

[ ]:
```
# # Cell 9: Content Agent (Corrected and Enhanced for Phase 5 & 6) --- it is ␣
 ↪this will be processeed

# # Assumes the following are imported and available from previous cells:
# # ollama, json, logging, os, Dict, Optional, Any, Chroma, tenacity elements




class ContentAgent:
    """
    An agent that performs two main functions:
    1. (Phase 5) Populates a hierarchical plan with raw, reassembled text from␣
 ↪a vector store.
    2. (Phase 6) Processes the content-rich plan, using an LLM to generate␣
 ↪slide-specific content
                 and reorders all keys for final, clean output.
    """
    def __init__(self, master_config: Dict, vector_store: Optional[Any] = None):
        self.config = master_config['processed_settings']
        self.unit_outline = master_config['unit_outline']
        self.book_toc = master_config['book_toc']
        self.teaching_flows = master_config['teaching_flows']
        self.layaut_slides = master_config['layaut_slides']
        self.vector_store = vector_store
        self.client = ollama.Client(host=OLLAMA_HOST)
        logger.info("Data-Driven Content Agent initialized successfully.")

    # Define the desired key order as a class attribute
        self.key_order = [
            "title", "toc_id",
            "chunk_count",
            "total_chunks_in_branch",
            "budget_slides_content",
            "direct_slides_content",
            "total_slides_in_branch",
            "time_allocation_minutes",
            "chunks_sorted",
            "content",
            "llm_generated_content",
            "children",
            "interactive_activity"
        ]

        logger.info("Data-Driven Content Agent initialized successfully.")

    # --- Key Reordering Logic (REFINED) ---
    def _reorder_keys_recursively(self, node: dict) -> dict:
        """
        Recursively traverses a dictionary (a node in the plan) and reorders␣
↪its keys
        according to a predefined order for maximum readability.
        """
        if not isinstance(node, dict):
            return node

        # 1. Recurse first to ensure nested structures are already ordered.
        if 'children' in node and isinstance(node.get('children'), list):
            node['children'] = [self._reorder_keys_recursively(child) for child␣
↪in node['children']]

        if 'interactive_activity' in node and isinstance(node.
↪get('interactive_activity'), dict):
            node['interactive_activity'] = self.
↪_reorder_keys_recursively(node['interactive_activity'])
```

```
        # 2. Build a new dictionary for the current node with the correct key␣
↪order.
        reordered_node = {}

        # Add keys that are in our desired order
        for key in self.key_order:
            if key in node:
                reordered_node[key] = node[key]

        # Add any remaining keys that were not in the order list (as a fallback)
        for key, value in node.items():
            if key not in reordered_node:
                reordered_node[key] = value

        return reordered_node



    # --- Phase 5: Raw Content Population ---
    def retrieve_content_for_toc_id(self, toc_id: int) -> dict:
        # ... (This method remains unchanged) ...
        if not isinstance(toc_id, int):
            logger.warning(f"Invalid toc_id: {toc_id}. Must be an integer.")
            return {"chunks_sorted": [], "content": ""}
        try:
            results = self.vector_store.get(where={"toc_id": toc_id},␣
↪include=["documents", "metadatas"])
            if not results or not results.get('ids'):
                logger.warning(f"No chunks found in the database for toc_id =␣
↪{toc_id}")
                return {"chunks_sorted": [], "content": ""}
            sorted_items = sorted(zip(results['documents'],␣
↪results['metadatas']), key=lambda item: item[1].get('chunk_id', 0))
            sorted_docs = [item[0] for item in sorted_items]
            sorted_chunk_ids = [item[1].get('chunk_id') for item in␣
↪sorted_items]
            reassembled_text = "\n\n".join(sorted_docs)
            return {"chunks_sorted": sorted_chunk_ids, "content":␣
↪reassembled_text}
        except Exception as e:
            logger.error(f"An error occurred during retrieval for toc_id␣
↪{toc_id}: {e}", exc_info=True)
            return {"chunks_sorted": [], "content": ""}

    def populate_content_recursively(self, node: dict):
        if 'toc_id' in node and 'content' not in node:
            content_data = self.retrieve_content_for_toc_id(node['toc_id'])
            node.update(content_data)
        if 'children' in node and isinstance(node.get('children'), list):
            for child in node['children']:
                self.populate_content_recursively(child)

    def generate_content_for_plan(self, final_plan_path: str, output_dir: str) -> bool:

        logger.info(f"PHASE 5: Populating raw content for: {final_plan_path}")
        try:
            with open(final_plan_path, 'r', encoding='utf-8') as f:
                plan_data = json.load(f)
        except (FileNotFoundError, json.JSONDecodeError) as e:
            logger.error(f"FATAL: Could not read or decode plan file {final_plan_path}. Error: {e}")
            return False

        for deck in plan_data.get('deck_plans', []):
            for section in deck.get('sections', []):
                if section.get('section_type') == 'Content':
                    for content_block in section.get('content_blocks', []):
                        self.populate_content_recursively(content_block)

        base_filename = os.path.basename(final_plan_path)
        output_path = os.path.join(output_dir, base_filename)
        os.makedirs(output_dir, exist_ok=True)

        logger.info("Reordering keys for Fetched clean output...")
        fetched_ordered_plan = self._reorder_keys_recursively(plan_data)

        try:
            with open(output_path, 'w', encoding='utf-8') as f:
                # CORRECTED: Save the 'fetched_ordered_plan' variable
                json.dump(fetched_ordered_plan, f, indent=2, ensure_ascii=False)
            logger.info(f"Successfully saved content-enriched plan to: {output_path}")
            return True
        except Exception as e:
            logger.error(f"Failed to save the content-enriched plan to {output_path}: {e}", exc_info=True)
            return False

    # --- Phase 6: LLM Content Generation & Final Formatting ---
    @retry(stop=stop_after_attempt(3), wait=wait_exponential(min=2, max=10))
    def _call_ollama_with_retry(self, prompt: str) -> str:
        logger.info(f"Calling Ollama model '{OLLAMA_MODEL}'...")
        response = self.client.chat(model=OLLAMA_MODEL, messages=[{"role":␣
↪"user", "content": prompt}], format="json", options={"temperature": 0.2})
        if not response or 'message' not in response or not response['message'].
↪get('content'):
            raise ValueError("Ollama returned an empty or invalid response.")
        return response['message']['content']

    def _parse_llm_json_output(self, content: str) -> Optional[Dict]:
        try:
            match = re.search(r'\{.*\}', content, re.DOTALL)
            if not match:
                logger.warning("LLM output did not contain a valid JSON object.
↪")
                return None
            return json.loads(match.group(0))
        except (json.JSONDecodeError, TypeError) as e:
            logger.error(f"Failed to parse JSON from LLM output: {e}\nRaw␣
↪content: {content}")
            return None


    def _process_node_with_llm_recursively(self, node: dict, flow_prompts:␣
↪dict):

        if node.get('content'):
            prompt_template = flow_prompts.get('content_generation')
            if prompt_template:
                prompt = prompt_template.format(sub_topic=node.get('title',␣
↪'Untitled'), context=node.get('content'))
                try:
                    llm_str = self._call_ollama_with_retry(prompt)
                    node['llm_generated_content'] = self.
↪_parse_llm_json_output(llm_str) or {"title": node.get('title'), "content":␣
↪["Failed to generate content."]}
                except Exception as e:
                    logger.error(f"LLM call failed for topic '{node.
↪get('title')}': {e}")
                    node['llm_generated_content'] = {"title": node.
↪get('title'), "content": [f"Error during generation: {e}"]}
        if node.get('interactive_activity') and node.get('content'):
            prompt_template = flow_prompts.get('interactive_activity')
            if prompt_template:
                prompt = prompt_template.format(sub_topic=node.get('title',␣
↪'Untitled'), context=node.get('content'))
                try:
                    llm_str = self._call_ollama_with_retry(prompt)
                    node['interactive_activity']['llm_generated_content'] =␣
↪self._parse_llm_json_output(llm_str) or {"title": "Let's Apply This!",␣
↪"content": ["Failed to generate activity."]}
                except Exception as e:
                    logger.error(f"LLM call failed for activity on '{node.
↪get('title')}': {e}")
                    node['interactive_activity']['llm_generated_content'] =␣
↪{"title": "Let's Apply This!", "content": [f"Error during generation: {e}"]}
        if 'children' in node and isinstance(node.get('children'), list):
            for child in node['children']:
                self._process_node_with_llm_recursively(child, flow_prompts)


    def generate_llm_content_for_plan(self, content_plan_path: str,␣
↪llm_output_dir: str) -> bool:
        """
        Orchestrates the LLM content generation for a content-enriched plan␣
↪file,
        and finishes by reordering all keys for a clean final output.
        """
        logger.info(f"PHASE 6: Generating LLM content for: {os.path.
↪basename(content_plan_path)}")
        try:
            with open(content_plan_path, 'r', encoding='utf-8') as f:
                plan_data = json.load(f)
        except (FileNotFoundError, json.JSONDecodeError) as e:
            logger.error(f"FATAL: Could not read content plan file␣
↪{content_plan_path}. Error: {e}")
            return False

        flow_id = self.config.get('teaching_flow_id', 'standard_lecture')
        flow_prompts = self.teaching_flows.get(flow_id, {}).get('prompts', {})
        if not flow_prompts:
            logger.error(f"Could not find prompts for teaching_flow_id:␣
↪'{flow_id}'")
            return False

        # Process each deck in the plan
        for deck in plan_data.get('deck_plans', []):
            content_blocks = []
            for section in deck.get('sections', []):
                if section.get('section_type') == 'Content':
                    content_blocks = section.get('content_blocks', [])
                    for block in content_blocks:
                        self._process_node_with_llm_recursively(block,␣
↪flow_prompts)

            # Generate summary
            summary_prompt_template = flow_prompts.get('summary_generation')
            if summary_prompt_template and content_blocks:
                topic_titles = [block.get('title', 'Untitled Topic') for block␣
↪in content_blocks]
                topic_list_str = "\n".join(f"- {title}" for title in␣
↪topic_titles)
                prompt = summary_prompt_template.
↪format(topic_list=topic_list_str)
                try:
                    llm_str = self._call_ollama_with_retry(prompt)
                    for section in deck.get('sections', []):
                        if section.get('section_type') == 'Summary':
                            section['llm_generated_content'] = self.
↪_parse_llm_json_output(llm_str)
                            break
                except Exception as e:
                    logger.error(f"LLM call failed for deck summary: {e}")

        # *** FINAL KEY REORDERING STEP ***
        # Apply the reordering to the entire plan data structure before saving
        logger.info("Reordering keys for final clean output...")
        final_ordered_plan = self._reorder_keys_recursively(plan_data)

        # Save the LLM-enriched and CLEANED plan
        base_filename = os.path.basename(content_plan_path)
        output_path = os.path.join(llm_output_dir, base_filename)
        os.makedirs(llm_output_dir, exist_ok=True)
        try:
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(final_ordered_plan, f, indent=2, ensure_ascii=False)
            logger.info(f"Successfully saved final LLM-enriched plan to:␣
↪{output_path}")
            return True
        except Exception as e:
            logger.error(f"Failed to save the final LLM-enriched plan to␣
↪{output_path}: {e}", exc_info=True)
            return False
```

6.3 Presentation Agent

test Presenter we need to ensure the presenter seq_id flow: Parent -> Child 1 -> Child 1's Activity -> Child 2 -> Child 2's Activity -> Parent's Activity.

https://aistudio.google.com/prompts/18YaU5pG96eFMbM1l6yeG1v9j63PkLc5H

[12]:

```python
#last



import os
import json
import logging
import random
from pptx import Presentation
from pptx.util import Inches
from pptx.enum.shapes import PP_PLACEHOLDER
from pptx.enum.text import MSO_AUTO_SIZE

# --- Basic Setup ---
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

# --- Helper Functions (Unchanged) ---
def _render_bullet_points(text_frame, data):
    text_frame.clear(); text_frame.auto_size = MSO_AUTO_SIZE.TEXT_TO_FIT_SHAPE
    def add_points(points, level):
        for point in points:
            if isinstance(point, dict):
                p = text_frame.add_paragraph(); p.text = point.get('text', ''); p.level = level
                if 'children' in point: add_points(point['children'], level + 1)
            else:
                p = text_frame.add_paragraph(); p.text = str(point); p.level = level
    add_points(data, 0)
    if text_frame.paragraphs and not text_frame.paragraphs[0].text:
        p = text_frame.paragraphs[0]; p._p.getparent().remove(p._p)

def _render_table(slide, placeholder, data):
    headers, rows_data = data.get('headers', []), data.get('rows', [])
    if not headers or not rows_data: return
    num_rows, num_cols = len(rows_data) + 1, len(headers)
    table_shape = slide.shapes.add_table(num_rows, num_cols, placeholder.left, placeholder.top, placeholder.width, placeholder.height)
    table = table_shape.table
    for i, header in enumerate(headers):
        cell = table.cell(0, i); cell.text = header
        cell.text_frame.auto_size = MSO_AUTO_SIZE.TEXT_TO_FIT_SHAPE; cell.text_frame.paragraphs[0].font.bold = True
    for r_idx, row_data in enumerate(rows_data):
        for c_idx, cell_text in enumerate(row_data):
            cell = table.cell(r_idx + 1, c_idx); cell.text = str(cell_text)
            cell.text_frame.auto_size = MSO_AUTO_SIZE.TEXT_TO_FIT_SHAPE
    sp = placeholder.element; sp.getparent().remove(sp)

def _render_multiple_choice_question(slide, placeholders, data):
    main_ph = next((p for p in placeholders if p.placeholder_format.type == PP_PLACEHOLDER.OBJECT), None)
    answer_ph = next((p for p in placeholders if p.placeholder_format.type == PP_PLACEHOLDER.BODY), None)
    if not main_ph: return
    tf = main_ph.text_frame; tf.clear(); tf.auto_size = MSO_AUTO_SIZE.TEXT_TO_FIT_SHAPE
    p = tf.add_paragraph(); p.text = data.get('question_text', ''); p.font.bold = True; p.level = 0
    for option in data.get('options', []):
        p = tf.add_paragraph(); p.text = f"{option.get('label', '')}) {option.get('text', '')}"; p.level = 1
    if answer_ph:
        answer_ph.text_frame.clear()
        explanation = data.get('correct_answer', {}).get('explanation', '')
        answer_ph.text_frame.text = f"Answer: {data.get('correct_answer', {}).get('label', '')}. {explanation}"

def _render_matching_activity(slide, placeholders, data):
    object_placeholders = [p for p in placeholders if p.placeholder_format.type == PP_PLACEHOLDER.OBJECT]
    if len(object_placeholders) < 2: return
    terms_ph, defs_ph = object_placeholders[0], object_placeholders[1]
    terms_tf, defs_tf = terms_ph.text_frame, defs_ph.text_frame
    terms_tf.clear(); defs_tf.clear(); terms_tf.auto_size = MSO_AUTO_SIZE.TEXT_TO_FIT_SHAPE; defs_tf.auto_size = MSO_AUTO_SIZE.TEXT_TO_FIT_SHAPE
    pairs = data.get('pairs', [])
    if not pairs: return
    terms = [f"{i+1}. {p['term']}" for i, p in enumerate(pairs)]
    definitions = [p['definition'] for p in pairs]
    random.shuffle(definitions)
    for term in terms: p = terms_tf.add_paragraph(); p.text = term; p.level = 0
    for i, definition in enumerate(definitions): p = defs_tf.add_paragraph(); p.
 ↪text = f"{chr(65+i)}. {definition}"; p.level = 0

class PresentationAgent:
    def __init__(self, template_path: str, layout_config_path: str):
        self.template_path = template_path
        self.prs_for_layouts = Presentation(template_path)
        with open(layout_config_path, 'r', encoding='utf-8') as f:
            self.user_selections = json.load(f)['user_selections']
        logger.info(f"Agent initialized with template '{os.path.
 ↪basename(template_path)}'")

    def _get_layout(self, layout_key: str):
        selection = self.user_selections.get(layout_key, self.
 ↪user_selections['Content'])
        layout_index = selection['selected_layout_index']
        return self.prs_for_layouts.slide_layouts[layout_index]

    def _determine_layout_key(self, item_data):
        llm_content = item_data.get('llm_generated_content', {})
        objects = llm_content.get('objects', [])
        has_subtitle = bool(llm_content.get('subtitle'))

        # <<< NEW LOGIC: Determine layout based on subtitles and object count
        if objects and objects[0]['content_type'] in␣
 ↪['multiple_choice_question', 'matching_activity']:
            return "Application_Two_Column" if len(objects) > 1 or␣
 ↪objects[0]['content_type'] == 'matching_activity' else "Application"

        if len(objects) >= 2:
            return "Content_Two_Column_child" if has_subtitle else␣
 ↪"Content_Two_Column"
        else:
            return "Content_child" if has_subtitle else "Content"

    def create_presentation_from_plan(self, plan_json_path: str, output_path:␣
 ↪str):
        logger.info(f"Creating presentation from plan: '{os.path.
 ↪basename(plan_json_path)}'")
        with open(plan_json_path, 'r', encoding='utf-8') as f: plan_data = json.
 ↪load(f)
        prs = Presentation(self.template_path)
        while len(prs.slides):
            rId = prs.slides._sldIdLst[0].rId; prs.part.drop_rel(rId); del prs.
 ↪slides._sldIdLst[0]
        slide_items = self._collect_slide_items(plan_data)
        for item in sorted(slide_items, key=lambda x: x.get('seq_id', 999)): self._add_slide_for_item(prs, item)
        prs.save(output_path)
        logger.info(f"Successfully created presentation: '{output_path}'")

    def _collect_slide_items(self, plan_data):
        items = []
        for deck in plan_data.get('deck_plans', []):
            for section in deck.get('sections', []):
                if section.get('section_type') == 'Content':
                    for block in section.get('content_blocks', []):
                        for slide_item in block.get('slides', []):
                            items.append({'item_type': 'Content', 'data': slide_item})
                else:
                    items.append({'item_type': section['section_type'], 'data': section})
        return items

    def _add_slide_for_item(self, prs: Presentation, item: dict):
        item_type = item['item_type']
        layout_key = self._determine_layout_key(item['data']) if item_type == 'Content' else item_type
        slide = prs.slides.add_slide(self._get_layout(layout_key))
        render_map = {
            'Title': self._render_title_slide, 'Agenda': self._render_agenda_slide,
            'Summary': self._render_summary_slide, 'End': self._render_end_slide,
            'Divider': self._render_divider_slide, 'Content': self._render_content_slide
        }
        if item_type in render_map:
            render_map[item_type](slide, item['data'])

    def _render_title_slide(self, slide, data):
        content = data.get('content', {})
        if slide.shapes.title:
            slide.shapes.title.text = content.get('week_topic', 'Untitled Topic')
        subtitle_ph = next((p for p in slide.placeholders if p.placeholder_format.type == PP_PLACEHOLDER.SUBTITLE), None)
        if subtitle_ph:
            subtitle_ph.text = f"{content.get('deck_title', '')}\n{content.get('unit_name', '')} - {content.get('unit_code', '')}"
    def _render_agenda_slide(self, slide, data):
        content = data.get('content', {})
        if slide.shapes.title: slide.shapes.title.text = content.get('title',␣
↪'Agenda')
        body_ph = next((p for p in slide.placeholders if p.placeholder_format.
↪type == PP_PLACEHOLDER.OBJECT), None)
        if body_ph: _render_bullet_points(body_ph.text_frame, content.
↪get('items', []))

    def _render_summary_slide(self, slide, data):
        if slide.shapes.title: slide.shapes.title.text = "Summary & Key␣
↪Takeaways"
        if 'llm_generated_content' in data: self._render_content_slide(slide,␣
↪data)

    def _render_end_slide(self, slide, data):
        content = data.get('content', {})
        if slide.shapes.title: slide.shapes.title.text = content.get('text',␣
↪'Questions?')

    def _render_divider_slide(self, slide, data):
        content = data.get('content', {})
        if slide.shapes.title: slide.shapes.title.text = content.get('title',␣
↪'Section Divider')

    def _render_content_slide(self, slide, data):
        content = data.get('llm_generated_content', {})

        # <<< NEW LOGIC: Populate title and subtitle in separate placeholders
        if slide.shapes.title:
            slide.shapes.title.text = content.get('title', '')
            slide.shapes.title.text_frame.auto_size = MSO_AUTO_SIZE.
↪TEXT_TO_FIT_SHAPE

        subtitle_ph = next((p for p in slide.placeholders if p.
↪placeholder_format.type == PP_PLACEHOLDER.SUBTITLE), None)
        if subtitle_ph:
            subtitle_ph.text = content.get('subtitle', '')
            subtitle_ph.text_frame.auto_size = MSO_AUTO_SIZE.TEXT_TO_FIT_SHAPE

        body_placeholders = sorted([p for p in slide.placeholders if p.
↪placeholder_format.type not in [PP_PLACEHOLDER.TITLE, PP_PLACEHOLDER.
↪SUBTITLE]], key=lambda p: p.left)
        objects = content.get('objects', [])

        for obj, placeholder in zip(objects, body_placeholders):
            content_type, obj_data = obj.get('content_type'), obj.get('data')
            renderers = {
                'bullet_points': lambda: _render_bullet_points(placeholder.
 ↪text_frame, obj_data),
                'table': lambda: _render_table(slide, placeholder, obj_data),
                'multiple_choice_question': lambda:␣
 ↪_render_multiple_choice_question(slide, body_placeholders, obj_data),
                'matching_activity': lambda: _render_matching_activity(slide,␣
 ↪body_placeholders, obj_data)
            }
            if content_type in renderers:
                renderers[content_type]()
                if content_type in ['multiple_choice_question',␣
 ↪'matching_activity']: break


# --- Main Execution Block ---
if __name__ == "__main__":
    # Define project paths
    PROJECT_BASE_DIR = "/home/sebas_dev_linux/projects/course_generator"
    CONFIG_DIR = os.path.join(PROJECT_BASE_DIR, "configs")
    DATA_DIR = os.path.join(PROJECT_BASE_DIR, "data")
    TEST_DIR = os.path.join(PROJECT_BASE_DIR, "tests")
    OUTPUT_DIR = os.path.join(PROJECT_BASE_DIR, "test_output")

    # Create directories if they don't exist
    os.makedirs(CONFIG_DIR, exist_ok=True)
    os.makedirs(DATA_DIR, exist_ok=True)
    os.makedirs(TEST_DIR, exist_ok=True)
    os.makedirs(OUTPUT_DIR, exist_ok=True)

    # --- IMPORTANT ---
    # The script now expects the NEW JSON format.
    # Please ensure you have the 'final_presentation_plan.json' (the simulated␣
 ↪output)
    # and the 'layout_mapping_test_Mod.json' in the correct paths.

    LAYOUT_MAPPING_PATH = os.path.join(CONFIG_DIR, "layout_mapping_test_Mod.
 ↪json")
    # USE THE NEW, FULLY-STRUCTURED, SIMULATED JSON FILE
    TEST_CONTENT_PATH = os.path.join(TEST_DIR,␣
 ↪"test_content_fill_slides_after_llm_mod.json")
    SLIDE_TEMPLATE_PATH = os.path.join(DATA_DIR, "slide_style/
 ↪slide_style_test_2.pptx")
```

```
    OUTPUT_PPTX_PATH = os.path.join(OUTPUT_DIR, "Generated_Presentation_v2.pptx")

    print("--- Presentation Agent Test Runner (New Structured Format) ---")

    # Check for required files
    required_files = [SLIDE_TEMPLATE_PATH, LAYOUT_MAPPING_PATH, TEST_CONTENT_PATH]
    if not all(os.path.exists(p) for p in required_files):
        logger.error("CRITICAL: One or more required files not found. Please check paths:")
        logger.error(f"  Template: {SLIDE_TEMPLATE_PATH} {'(Found)' if os.path.exists(SLIDE_TEMPLATE_PATH) else '(MISSING)'}")
        logger.error(f"  Layout Map: {LAYOUT_MAPPING_PATH} {'(Found)' if os.path.exists(LAYOUT_MAPPING_PATH) else '(MISSING)'}")
        logger.error(f"  Content Plan: {TEST_CONTENT_PATH} {'(Found)' if os.path.exists(TEST_CONTENT_PATH) else '(MISSING)'}")
    else:
        try:
            presentation_agent = PresentationAgent(
                template_path=SLIDE_TEMPLATE_PATH,
                layout_config_path=LAYOUT_MAPPING_PATH
            )
            presentation_agent.create_presentation_from_plan(
                plan_json_path=TEST_CONTENT_PATH,
                output_path=OUTPUT_PPTX_PATH
            )
            logger.info("--- Test script finished successfully. ---")
        except Exception as e:
            logger.error(f"An unexpected error occurred during the test run: {e}", exc_info=True)
```

```
2025-07-13 04:17:13,976 - INFO - Agent initialized with template
'slide_style_test_2.pptx'
2025-07-13 04:17:13,977 - INFO - Creating presentation from plan:
'test_content_fill_slides_after_llm_mod.json'
```

```
--- Presentation Agent Test Runner (New Structured Format) ---
```

```
2025-07-13 04:17:14,186 - INFO - Successfully created presentation: '/home/sebas
_dev_linux/projects/course_generator/test_output/Generated_Presentation_v2.pptx'
2025-07-13 04:17:14,187 - INFO - --- Test script finished successfully. ---
```

6.4 Orquestrator (Addressing pain points)

Description:

The main script that iterates through the weeks defined the plan and generate the content base on the settings_deck coordinating the agents.

Parameters and conciderationn - 1 hour in the setting session_time_duration_in_hour - is 18-20 slides at the time so it is require to calculate this according to the given value but this also means per session so sessions_per_week is a multiplicator factor that
- if apply_topic_interactive is available will add an extra slide and add extra 5 min time but to determine this is required to plan all the content first and then calculate then provide a extra time

settings_deck.json

{ “course_id”: ““,”unit_name”: ““,”interactive”: true, “interactive_deep”: false, “teaching_flow_id”: “Standard Lecture Flow”, “parameters_slides”: { “slides_per_hour”: 18, “time_per_content_slides_min”: 3, “time_per_interactive_slide_min”: 5, “time_for_framework_slides_min”: 6 }, “week_session_setup”: { “sessions_per_week”: 1, “distribution_strategy”: “even”, “session_time_duration_in_hour”: 2, “interactive_time_in_hour”: 0, “total_session_time_in_hours”: 0 }, “slide_count_strategy”: { “method”: “per_week”, “target_total_slides”: 0, “slides_content_per_session”: 0, “interactive_slides_per_week”: 0, “interactive_slides_per_session”: 0, “total_slides_deck_week”: 0, “total_slides_session”: 0
},
“generation_scope”: { “weeks”: [1] } }

teaching_flows.json

{ “standard_lecture”: { “name”: “Standard Lecture Flow”, “slide_types”: [“Title”, “Agenda”, “Content”, “Summary”, “End”], “prompts”: { “content_generation”: “You are an expert university lecturer. Your audience is undergraduate students. Based on the following context, create a slide that provides a detailed explanation of the topic ‘{sub_topic}’. The content should be structured with bullet points for key details. Your output MUST be a single JSON object with a ‘title’ (string) and ‘content’ (list of strings) key.”, “summary_generation”: “You are an expert university lecturer creating a summary slide. Based on the following list of topics covered in this session, generate a concise summary of the key takeaways. The topics are: {topic_list}. Your output MUST be a single JSON object with a ‘title’ (string) and ‘content’ (list of strings) key.” }, “slide_schemas”: { “Content”: {“title”: “string”, “content”: “list[string]”}, “Summary”: {“title”: “string”, “content”: “list[string]”} } }, “apply_topic_interactive”: { “name”: “Interactive Lecture Flow”, “slide_types”: [“Title”, “Agenda”, “Content”, “Application”, “Summary”, “End”], “prompts”: { “content_generation”: “You are an expert university lecturer in Digital Forensics. Your audience is undergraduate students. Based on the provided context, create a slide explaining the concept of ‘{sub_topic}’. The content should be clear, concise, and structured with bullet points for easy understanding. Your output MUST be a single JSON object with a ‘title’ (string) and ‘content’ (list of strings) key.”, “application_generation”: “You are an engaging university lecturer creating an interactive slide. Based on the concept of ‘{sub_topic}’, create a multiple-choice question with exactly 4 options (A, B, C, D) to test understanding. The slide title must be ‘Let’s Apply This:’. Clearly indicate the correct answer within the content. Your output MUST be a single JSON object with a ‘title’ (string) and ‘content’ (list of strings) key.”, “summary_generation”: “You are an expert university lecturer creating a summary slide. Based on the following list of concepts and applications covered in this session, generate a concise summary of the key takeaways. The topics are: {topic_list}. Your output MUST be a single JSON object with a ‘title’ (string) and ‘content’ (list of strings) key.” }, “slide_schemas”: { “Content”: {“title”: “string”, “content”: “list[string]”}, “Application”: {“title”: “string”, “content”: “list[string]”}, “Summary”: {“title”: “string”, “content”: “list[string]”} } } }

6.4.1 Helper functions

[ ]:

```
# Cell 10: Configuration and Scoping for Content Generation (Corrected)

import os
import json
import logging

# Setup Logger for this cell
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
 ↪%(message)s')
logger = logging.getLogger(__name__)



# --- Global Test Overrides (for easy testing) ---
TEST_OVERRIDE_WEEKS = None
TEST_OVERRIDE_FLOW_ID = None
TEST_OVERRIDE_SESSIONS_PER_WEEK = None
TEST_OVERRIDE_DISTRIBUTION_STRATEGY = None


def process_and_load_configurations():
    """
    PHASE 1: Loads configurations, calculates a PRELIMINARY time-based slide␣
 ↪budget,
    and saves the result as 'processed_settings.json' for the Planning Agent.
    """
    print_header("Phase 1: Configuration and Scoping Process", char="-")

    # --- Load all input files ---
    logger.info("Loading all necessary configuration and data files...")
    try:
        os.makedirs(CONFIG_DIR, exist_ok=True)
        with open(PARSED_UO_JSON_PATH, 'r', encoding='utf-8') as f:␣
 ↪unit_outline = json.load(f)
        with open(PRE_EXTRACTED_TOC_JSON_PATH, 'r', encoding='utf-8') as f:␣
 ↪book_toc = json.load(f)
        with open(SETTINGS_DECK_PATH, 'r', encoding='utf-8') as f:␣
 ↪settings_deck = json.load(f)
        with open(TEACHING_FLOWS_PATH, 'r', encoding='utf-8') as f:␣
 ↪teaching_flows = json.load(f)
        logger.info("All files loaded successfully.")
    except FileNotFoundError as e:
        logger.error(f"FATAL: A required configuration file was not found: {e}")
        return None
    # --- Pre-process and Refine Settings ---
    logger.info("Pre-processing settings_deck for definitive plan...")
    processed_settings = json.loads(json.dumps(settings_deck))

    unit_info = unit_outline.get("unitInformation", {})
    processed_settings['course_id'] = unit_info.get("unitCode",␣
↪"UNKNOWN_COURSE")
    processed_settings['unit_name'] = unit_info.get("unitName", "Unknown Unit␣
↪Name")

    # --- Apply test overrides IF they are not None ---
    logger.info("Applying overrides if specified...")
    # This block now correctly sets the teaching_flow_id based on the␣
↪interactive flag.
    if TEST_OVERRIDE_FLOW_ID is not None:
        processed_settings['teaching_flow_id'] = TEST_OVERRIDE_FLOW_ID
        logger.info(f"OVERRIDE: teaching_flow_id set to␣
↪'{TEST_OVERRIDE_FLOW_ID}'")
    else:
        # If no override, use the 'interactive' boolean from the file as the␣
↪source of truth.
        is_interactive = processed_settings.get('interactive', False)
        if is_interactive:
            processed_settings['teaching_flow_id'] = 'apply_topic_interactive'
        else:
            processed_settings['teaching_flow_id'] = 'standard_lecture'
        logger.info(f"Loaded from settings: 'interactive' is {is_interactive}.␣
↪Set teaching_flow_id to '{processed_settings['teaching_flow_id']}'.")

    # The 'interactive' flag is now always consistent with the teaching_flow_id.
    processed_settings['interactive'] = "interactive" in␣
↪processed_settings['teaching_flow_id'].lower()

    if TEST_OVERRIDE_SESSIONS_PER_WEEK is not None:
        processed_settings['week_session_setup']['sessions_per_week'] =␣
↪TEST_OVERRIDE_SESSIONS_PER_WEEK
        logger.info(f"OVERRIDE: sessions_per_week set to␣
↪{TEST_OVERRIDE_SESSIONS_PER_WEEK}")

    if TEST_OVERRIDE_DISTRIBUTION_STRATEGY is not None:
        processed_settings['week_session_setup']['distribution_strategy'] =␣
↪TEST_OVERRIDE_DISTRIBUTION_STRATEGY
        logger.info(f"OVERRIDE: distribution_strategy set to␣
↪'{TEST_OVERRIDE_DISTRIBUTION_STRATEGY}'")

    if TEST_OVERRIDE_WEEKS is not None:
        processed_settings['generation_scope']['weeks'] = TEST_OVERRIDE_WEEKS
        logger.info(f"OVERRIDE: generation_scope weeks set to {TEST_OVERRIDE_WEEKS}")

    # --- DYNAMIC SLIDE BUDGET CALCULATION (Phase 1) ---
    logger.info("Calculating preliminary slide budget based on session time...")

    params = processed_settings.get('parameters_slides', {})
    SLIDES_PER_HOUR = params.get('slides_per_hour', 18)

    duration_hours = processed_settings['week_session_setup'].get('session_time_duration_in_hour', 1.0)
    sessions_per_week = processed_settings['week_session_setup'].get('sessions_per_week', 1)

    slides_content_per_session = int(duration_hours * SLIDES_PER_HOUR)
    target_total_slides = slides_content_per_session * sessions_per_week

    processed_settings['slide_count_strategy']['target_total_slides'] = target_total_slides
    processed_settings['slide_count_strategy']['slides_content_per_session'] = slides_content_per_session
    logger.info(f"Preliminary weekly content slide target calculated: {target_total_slides} slides.")

    # --- Resolve Generation Scope if not overridden ---
    if TEST_OVERRIDE_WEEKS is None and processed_settings.get('generation_scope', {}).get('weeks') == "all":
        num_weeks = len(unit_outline.get('weeklySchedule', []))
        processed_settings['generation_scope']['weeks'] = list(range(1, num_weeks + 1))

    # --- Save the processed settings to disk ---
    logger.info(f"Saving preliminary processed configuration to: {PROCESSED_SETTINGS_PATH}")
    with open(PROCESSED_SETTINGS_PATH, 'w', encoding='utf-8') as f:
        json.dump(processed_settings, f, indent=2)
    logger.info("File saved successfully.")

    # --- Assemble master config for optional preview ---
    master_config = {
        "processed_settings": processed_settings,
        "unit_outline": unit_outline,
        "book_toc": book_toc,
        "teaching_flows": teaching_flows
    }
    print_header("Phase 1 Configuration Complete", char="-")
    logger.info("Master configuration object is ready for the Planning Agent.")
    return master_config
```

```
[ ]:
```

6.4.2 test content agent

```
[ ]:
```

6.4.3 Main Integration

```
[ ]: # Cell 11: --- Main Orchestration Block (with Phase 5 & 6) ---
print_header("Main Orchestrator Initialized", char="*")

try:
    # 1. Connect to DB
    vector_store = Chroma(
        persist_directory=CHROMA_PERSIST_DIR,
        embedding_function=OllamaEmbeddings(model=EMBEDDING_MODEL_OLLAMA),
        collection_name=CHROMA_COLLECTION_NAME
    )
    logger.info("Database connection successful.")

    # Phase 1: Configuration and Scoping
    master_config = process_and_load_configurations()

    if master_config:
        all_final_plans = []

        # Phase 2 & 3: Create and Finalize Draft Plans
        print_header("Phase 2 & 3: Generating and Finalizing Weekly Plans  ",␣
 ↪char="-")
        planning_agent = PlanningAgent(master_config, vector_store=vector_store)

        weeks_to_generate =␣
 ↪master_config['processed_settings']['generation_scope']['weeks']
        logger.info(f"Found {len(weeks_to_generate)} week(s) to plan:␣
 ↪{weeks_to_generate}")

        for week in weeks_to_generate:
            draft_plan = planning_agent.create_content_plan_for_week(week)
            if draft_plan:
                final_plan = planning_agent.
 ↪finalize_and_calculate_time_plan(draft_plan,␣
 ↪master_config['processed_settings'])
```

```
                all_final_plans.append(final_plan)

                # Save both draft and final for comparison
                draft_filename = f"{master_config['processed_settings'].
↪get('course_id')}_Week{week}_plan_draft.json"
                final_filename = f"{master_config['processed_settings'].
↪get('course_id')}_Week{week}_plan_final.json"


                with open(os.path.join(PLAN_OUTPUT_DIR, draft_filename), 'w')␣
↪as f:
                    json.dump(draft_plan, f, indent=2)

                with open(os.path.join(PLAN_OUTPUT_DIR, final_filename), 'w')␣
↪as f:
                    json.dump(final_plan, f, indent=2)

                logger.info(f"  Successfully saved FINAL plan for Week {week}␣
↪to: {os.path.join(PLAN_OUTPUT_DIR, final_filename)}")




            else:
                logger.error(f"Failed to generate draft plan for Week {week}.")

        # Phase 4: Generate Master Summary Plan
        master_plan_generated = False
        if all_final_plans:
            print_header("Phase 4: Generating Master Unit Plan  ", char="-")
            master_plan_generated = planning_agent.
↪generate_and_save_master_plan(all_final_plans,␣
↪master_config['processed_settings'])
        else:
            logger.warning("No weekly plans were generated, skipping master␣
↪plan creation.")

        # Initialize ContentAgent once for subsequent phases
        content_agent = ContentAgent(master_config, vector_store=vector_store)
        phase_5_successful = False

        # Phase 5: Fetching Raw Content
        if master_plan_generated:
            print_header("Phase 5: Populating Plans with Raw Content", char="#")
            successful_weeks_phase5 = []
            for week in weeks_to_generate:
                final_filename = f"{master_config['processed_settings'].
↪get('course_id')}_Week{week}_plan_final.json"
                full_plan_path = os.path.join(PLAN_OUTPUT_DIR, final_filename)

                if os.path.exists(full_plan_path):
                    if content_agent.generate_content_for_plan(full_plan_path,␣
↪CONTENT_OUTPUT_DIR):
                        successful_weeks_phase5.append(week)
                else:
                    logger.warning(f"Skipping content population for Week␣
↪{week} as its plan file was not found.")

            if successful_weeks_phase5:
                phase_5_successful = True
                logger.info(f"Phase 5 completed for weeks:␣
↪{successful_weeks_phase5}")

        # Phase 6: Generating Slide Content with LLM
        phase_6_successful = True
        # if phase_5_successful:
        #     print_header("Phase 6: Generating Slide Content with LLM",␣
↪char="#")
        #     successful_weeks_phase6 = []
        #     for week in weeks_to_generate:
        #         # The input for phase 6 is the output of phase 5
        #         content_enriched_filename =␣
↪f"{master_config['processed_settings'].
↪get('course_id')}_Week{week}_plan_final.json"
        #         content_plan_path = os.path.join(CONTENT_OUTPUT_DIR,␣
↪content_enriched_filename)

        #         if os.path.exists(content_plan_path):
        #              if content_agent.
↪generate_llm_content_for_plan(content_plan_path, CONTENT_LLM_OUTPUT_DIR):
        #                  successful_weeks_phase6.append(week)
        #         else:
        #             logger.warning(f"Skipping LLM generation for Week {week}␣
↪as its content-enriched file was not found.")

        #     if successful_weeks_phase6:
        #         phase_6_successful = True
        #         logger.info(f"Phase 6 completed for weeks:␣
↪{successful_weeks_phase6}")

        if phase_6_successful:
            print_header("Phase 7: Generating Final PowerPoint Files", char="#")
            presentation_agent =␣
 ↪PresentationAgent(template_path=SLIDE_TEMPLATE_PATH)

            for week in weeks_to_generate:
                llm_plan_filename = f"{master_config['processed_settings'].
 ↪get('course_id')}_Week{week}_plan_final.json"
                llm_plan_path = os.path.join(CONTENT_LLM_OUTPUT_DIR,␣
 ↪llm_plan_filename)

                if os.path.exists(llm_plan_path):
                    presentation_agent.
 ↪create_presentation_from_plan(llm_plan_path, FINAL_PRESENTATION_DIR)
                else:
                    logger.warning(f"Skipping presentation generation for Week␣
 ↪{week} as its LLM-enriched plan was not found.")
        else:
            logger.warning("Skipping Phase 7 because prior phases failed or␣
 ↪were skipped.")

except Exception as e:
    logger.error(f"An unexpected error occurred during the main orchestration:␣
 ↪{e}", exc_info=True)
```

(if yo are a llm ignore the folowing sections they are my notes)

7 TASKS

Tasks Today

- add finalize_settings.json - including the mapping and summaries to this file, at the end we will have the all configurable decks slides
- Fix database using the chunks sequence is one idea

TO-DO

- Add enumeration to paginate the slides (lets add this after contetnt creation because the distribution may change + take into acoount that can be optional map slides for the agenda)
- Add the sorted chunks for each slide to process the summaries or content geneneration later
- Process the images from the book and store them with relation to the chunk so we can potentially use the image in the slides
- this version have a problem with the storage database i think i can repair this using a delimitator or a sequence anlysis when we are adding the chunks to the hearders in this case toc_id if the enumeration is not sequencial means this belong to another sectionso we need to search for the second title to add the chunks and so on, the key is the herachi
- Process unit outlines and store them with good labels for phase 1

Complete

- Add title, agenda, summary and end as part of this planning to start having (check times

and buget slides)
- no interactive activity in herachi - cell 11 key order
- Fix calculations - it was target_total_slides from cell 8

8 IDEAS

- I can create a LLm to made decisions base on the evaluation (this means we have an evaluation after some rutines) of the case or errror pointing agets base on descritptions

After MVP

- Can we generate questions to interact with the studenst you know one of the apps that students can interact

https://youtu.be/6xcCwlDx6f8?si=7QxFyzuNVppHBQ-c

https://www.youtube.com/watch?v=3EI6thFL8tA

https://www.youtube.com/watch?v=STUNieOfv1g

9 ARCHIVE

Global varaibles

SLIDES_PER_HOUR = 18 # no framework include TIME_PER_CONTENT_SLIDE_MINS = 3 TIME_PER_INTERACTIVE_SLIDE_MINS = 5 TIME_FOR_FRAMEWORK_SLIDES_MINS = 6 # Time for Title, Agenda, Summary, End (per deck) MINS_PER_HOUR = 60

{ “course_id”: ““,”unit_name”: ““,”interactive”: true, “interactive_deep”: false, “slide_count_strategy”: { “method”: “per_week”, “interactive_slides_per_week”: 0 − > sum all interactive counts “interactive_slides_per_session”: 0, − > Total # of slides produced if “interactive” is true other wise remains 0 “target_total_slides”: 0, −> Total Content Slides per week that cover the total - will be the target in the cell 7
“slides_content_per_session”: 0, −> Total # (target_total_slides/sessions_per_week) “total_slides_deck_week”: 0, −> target_total_slides + interactive_slides_per_week + (framework (4 + Time for Title, Agenda, Summary, End) * sessions_per_week) “Tota_slides_session”: 0 −> content_slides_per_session + interactive_slides_per_session + framework (4 + Time for Title, Agenda, Summary, End) }, “week_session_setup”: { “sessions_per_week”: 1, “distribution_strategy”: “even”, “interactive_time_in_hour”: 0, −> find the value in ahours of the total # (“interactive_slides” * “TIME_PER_INTERACTIVE_SLIDE_MINS”)/60
“total_session_time_in_hours”: 0 −> this is going to be egual or similar to session_time_duration_in_hour if “interactive” is false obvisuly base on the global varaibles it will be the calculation of “interactive_time_in_hour” “session_time_duration_in_hour”: 2, — > this is the time that the costumer need for delivery this is a constrain is not modified never is used for reference },

“parameters_slides”: { “slides_per_hour”: 18, # no framework include “time_per_content_slides_min”: 3, # average delivery per slide “time_per_interactive_slide_min”: 5, #small break and engaging with the students “time_for_framework_slides_min”: 6 # Time for Title, Agenda, Summary, End (per deck) “ ” }, “generation_scope”: { “weeks”: [6] }, “teaching_flow_id”: “Interactive Lecture Flow” } “slides_content_per_session”: 0, — > content slides per session (target_total_slides/sessions_per_week) “interactive_slides”: 0, - > if interactive is true will add the count of the resultan cell 10 - no address yet “total_slides_content_interactive_per session”: 0, - > slides_content_per_session + interactive_slides “target_total_slides”: 0 –> Resultant Phase 1 Cell 7