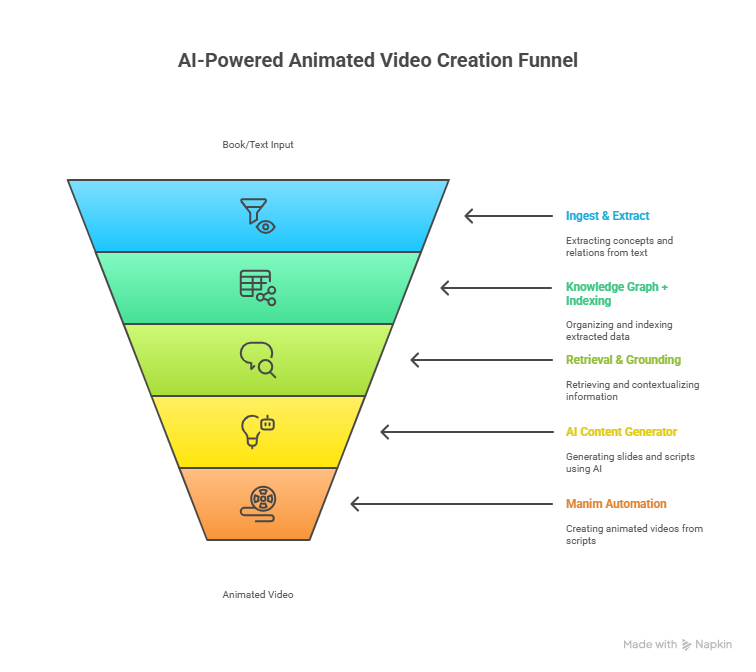
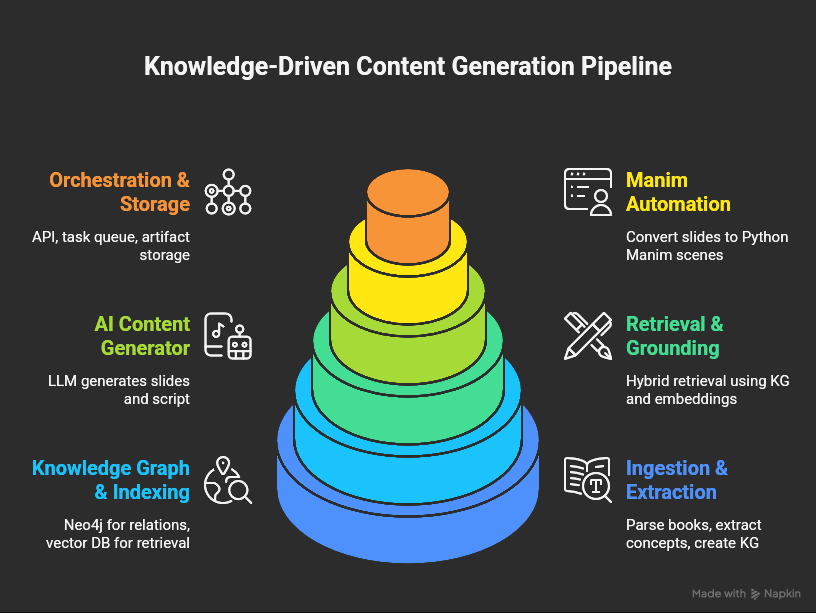
To Build an **AI-powered pipeline** that turns a structured knowledge graph of book content into fully automated **Manim** animation videos (slides + spoken script + rendered video).  
The flow of the project is as follows

The tech stack of the project with the following in mind will include:  
**Ingest & Extract** — parse books, extract concepts & relations, create KG + text corpus.  
**Knowledge Graph + Indexing** — Neo4j (or RDF) for relations + Vector DB for semantic retrieval.  
**Retrieval & Grounding** — hybrid retrieval (KG traversal + embedding search).  
**AI Content Generator** — LLM generates slides & script using RAG and templates.  
**Manim Automation** — convert formatted slides/scripts into Python Manim scenes, render to video.  
**Orchestration & Storage** — API, task queue, artifact storage, UI for students



**The diagram with the tech stack**

**Ingest & Extract  
Goal: To Convert book PDFs / EPUB / Markdown into structured KG nodes (Concept, Definition, Example, Equation, Diagram, Prerequisite).  
Steps I will take: OCR and PDF parsing (if needed).  
Heuristic chapter/section segmentation.  
Sentence-level embedding + clustering for candidate concepts.  
Named entity and relation extraction (rules + fine-tuned models).  
Human-in-the-loop verification for the initial dataset.**

The KG Schema example:

Node types:  
Concept {id, title, canonical\_text, examples[], difficulty, source\_refs[]}  
Definition {id, text, source\_ref}  
Example {id, text, inputs, outputs}  
Formula {id, latex, explanation}  
Diagram {id, svg\_or\_image\_ref}

Edges:  
(Concept)-[:HAS\_DEFINITION]->(Definition)  
(Concept)-[:HAS\_EXAMPLE]->(Example)  
(Concept)-[:PREREQUISITE]->(Concept)  
(Concept)-[:HAS\_DIAGRAM]->(Diagram)  
(Concept)-[:SOURCE]->(BookChapter)

And for the Ingest pseudo-code:

def ingest\_book(file\_path):

text\_blocks = parse\_pdf\_or\_epub(file\_path)

sections = segment\_sections(text\_blocks)

for sec in sections:

candidates = extract\_candidate\_concepts(sec)

for c in candidates:

canonical = normalize\_concept\_title(c)

triples = extract\_relations\_and\_triples(sec, c)

store\_in\_kg(canonical, triples, source=sec.ref)

index\_embeddings\_for\_text(sec.text, id=canonical.id)

**Retrieval & Grounding  
Goal:** Given a user request (e.g., “Explain eigenvectors”), find relevant KG node(s) and supporting passages.  
Hybrid retrieval algorithm (pseudo):  
def retrieve\_concept(query):

node = kg.find\_node\_by\_title(query)

if node:

return node, kg.get\_connected\_nodes(node)

candidates = vector\_db.search(query, top\_k=10)

scored = score\_candidates\_with\_kg\_proximity(candidates)

return select\_top(scored, k=3)

**AI Content Generator (Slides + Script)  
Goal: Generate a sequence of slides and an accompanying spoken script grounded to retrieved sources.  
Design decisions:  
Use structured prompting with a rigid output JSON schema to avoid model drift.  
Attach a context with retrieved passages (RAG) and explicit citations.  
Include a "verification pass" (LLM or deterministic checks) to ensure formulas are consistent (e.g., LaTeX syntax validity) and citations present.  
Provide slide templates: Title, Motivation, Intuition, Formal Definition, Example (worked), Visual/Diagram, Summary, Quiz.**

Prompt skeleton (conceptual):  
INSTRUCTIONS:

- Use the following retrieved sources (with IDs). Provide output as JSON: {slides: [...], script: "...", citations: [...]}

- Each slide: {title, bullet\_points, speaker\_notes, diagrams\_latex\_or\_svg}

- Keep slides to 6-10 for a short explainer.

CONTEXT:

- [passage1\_id] text...

- [passage2\_id] text...

USER\_REQUEST: "Explain X at Y level"

Generator pseudo-code:  
def generate\_slides\_and\_script(retrieved\_nodes, level="undergrad"):

context = compile\_context(retrieved\_nodes)

prompt = build\_prompt(context, level)

raw\_json = call\_llm(prompt)

parsed = validate\_and\_parse(raw\_json)

verified = run\_fact\_checker(parsed, context)

return verified

**Slide & Script Formatter  
Goal: Convert LLM output into render-ready slide objects and attach metadata.  
Slide object example:  
{**

**"slide\_id": "s1",**

**"title": "What is X?",**

**"layout": "title+bullets+image",**

**"elements": [**

**{"type":"text", "value":"Definition: ..."},**

**{"type":"bullet\_list", "items":["motivation", "intuition"]},**

**{"type":"diagram", "svg\_ref":"diag123.svg"}**

**],**

**"speaker\_notes": "Short script lines with timing 0:00-0:15",**

**"duration\_seconds": 15**

**}**

Formatter pseudo-code:  
**def format\_for\_manim(json\_slides):**

**scenes = []**

**for s in json\_slides:**

**scene = map\_slide\_to\_scene\_template(s)**

**scenes.append(scene)**

**return scenes**

**Manim Automation (core of the ask)  
Goal: Convert scene objects into Python Manim scenes, render, and stitch to an MP4.  
Key choices:  
Use Manim Community Edition (Python) because it’s scriptable and widely used.  
Provide templates per slide layout (title, bullets, diagram animation, code walkthrough, step-by-step math derivation).  
For domain-specific visuals (GIS), call external renderers (Mapbox snapshot + animated overlays) and embed them as video segments or images.**

Manim code generation pseudo-code:  
**MANIM\_TEMPLATE = """**

**from manim import \***

**class Scene\_{id}(Scene):**

**def construct(self):**

**title = Text("{title}").to\_edge(UP)**

**self.play(Write(title))**

**{body\_code}**

**self.wait({duration})**

**"""**

**def map\_slide\_to\_manim\_code(slide):**

**body\_code = ""**

**if slide.layout == "title+bullets":**

**body\_code += generate\_bullets\_code(slide.elements)**

**if slide.contains\_diagram:**

**body\_code += embed\_svg\_as\_mobject(slide.diagram\_ref)**

**return MANIM\_TEMPLATE.format(id=slide.slide\_id, title=escape(slide.title), body\_code=body\_code, duration=slide.duration\_seconds)**

**Rendering orchestration:  
Convert all slide scenes into Python files -> call manim -pql (or use programmatic API) in worker container -> concatenate audio (text-to-speech) with rendered video segments -> final mux.  
Audio / voice choice: Use TTS (Amazon Polly, Azure, or locally-run TTS). Provide SSML to emphasize math tokens where needed. Also allow human voice-over upload.**

**Backend Orchestration & APIs  
Components I choose:  
API: FastAPI (async, easy to build).  
Workers: Kubernetes Jobs or Celery w/ Redis.  
Storage: S3-compatible (videos, assets), Postgres for metadata.  
Indexing: Neo4j for KG; Milvus/Pinecone for vectors.  
Monitoring: Prometheus + Grafana; Sentry for errors.  
Auth: OAuth2 / API keys for controlled access.**

API example endpoints:  
POST /generate

body: {concept: "Eigenvectors", level: "undergrad", voice: "en-US-Wavenet-F"}

returns: {job\_id}

GET /status/{job\_id}

returns: {status, progress, preview\_url}

GET /download/{job\_id}

returns: video file or presigned URL

POST /feedback/{job\_id}

body: {rating, comments}

**Orchestration pseudo-code:**def handle\_request(request):

node = retrieve\_concept(request.concept)

slides\_script = generate\_slides\_and\_script(node, level=request.level)

scenes = format\_for\_manim(slides\_script)

job = enqueue\_render\_job(scenes, tts\_params=request.voice)

return job.id