KNOWY

Agentic Multimodal Architecture for Personal Knowledge Compilation, Epistemic Evolution, and Expert-Level Reasoning

Technical Design Document

Alberto Espinosa

January 16, 2026

Abstract

This document presents the comprehensive technical design for KNOWY, an agentic multimodal architecture designed to compile personal knowledge collections, support epistemic evolution, and enable expert-level reasoning. The system addresses the fundamental challenge of epistemic integrity in personal AI systems through a rigorously layered architecture that distinguishes between shallow operational agents and deep epistemic agents. KNOWY integrates advanced multimodal knowledge acquisition including structured video analysis, diagrammatic explanation generation, and a command-line interaction interface for local-first operation. This design specification details the system architecture, component implementations, data flows, algorithms, and deployment strategies necessary for realizing a robust epistemic infrastructure.

Contents

1 Executive Summary **4**
- 1.1 Vision and Scope . . . 4
- 1.2 Core Design Principles . . . 4
- 1.3 Key Innovations . . . 4

2 System Architecture **4**
- 2.1 Architectural Overview . . . 4
- 2.2 Architectural Guarantees . . . 6
- 2.3 Technology Stack . . . 6

3 Layer 0: Universal Ingestion **6**
- 3.1 Purpose and Epistemic Question . . . 6
- 3.2 Canonical Source Descriptor Schema . . . 7
- 3.3 Ingestion Pipeline Architecture . . . 8
- 3.4 Browser Extension Integration . . . 8
- 3.5 Format Detection Algorithm . . . 10
- 3.6 Storage Schema . . . 10

4 Layer 1: Exploration and Expansion **10**
- 4.1 Purpose and Epistemic Question . . . 10
- 4.2 Expansion Agent Architecture . . . 11
- 4.3 Knowledge Artifact Data Structure . . . 12
- 4.4 Reference Resolution Strategies . . . 13
- 4.5 Deduplication Algorithm . . . 13

5 Layer 2: Multimodal Acquisition and Scraping **14**
5.1 Purpose and Epistemic Question . . . 14
5.2 Acquisition Pipeline Dispatcher . . . 14
5.3 Video Analysis Pipeline Integration . . . 15
5.3.1 Video Analysis Architecture . . . 16
5.3.2 Shot Segmentation Algorithm . . . 17
5.3.3 Transcript-Visual Alignment . . . 17
5.3.4 Semantic Frame Analysis . . . 18
5.3.5 Multimodal Chunk Generation . . . 19
5.4 PDF Acquisition Pipeline . . . 20
5.5 Academic Paper Pipeline . . . 21

6 Layer 3: Multi-Resolution Knowledge Representation **22**
6.1 Purpose and Epistemic Question . . . 22
6.2 Four-Level Abstraction Hierarchy . . . 23
6.3 Representation Architecture . . . 23
6.4 Level 0: Raw Content Storage . . . 23
6.5 Level 1: Section-Level Representation . . . 24
6.6 Level 2: Document-Level Synthesis . . . 25
6.7 Level 3: Cross-Document Thematic Synthesis . . . 27
6.8 Dynamic Resolution Selection . . . 28

7 Layer 4: Classification and Ontology Construction **29**
7.1 Purpose and Epistemic Question . . . 29
7.2 Ontology Structure . . . 29
7.3 Topic Extraction and Clustering . . . 30
7.4 Redundancy Detection . . . 33

8 Layer 5: Expert Reasoning and Teaching **34**
8.1 Purpose and Epistemic Question . . . 34
8.2 Expert Reasoning Architecture . . . 34
8.3 Pedagogical Mode . . . 37

9 Layer 6: Diagrammatic Explanation **39**
9.1 Purpose and Epistemic Question . . . 39
9.2 Diagram Generation Architecture . . . 39
9.3 Concept Map Generation . . . 41
9.4 Mermaid Diagram Generation . . . 42

10 Layer 7: Deep External Research Agent **43**
10.1 Purpose and Epistemic Question . . . 43
10.2 DERA Architecture . . . 44
10.3 Research Trigger Detection . . . 47
10.4 Knowledge Versioning . . . 48

11 User Interaction Layer: CLI Interface **49**
11.1 CLI Architecture . . . 49
11.2 CLI Usage Examples . . . 50

12 System Integration and Orchestration **51**
12.1 LangGraph Orchestration . . . 51
12.2 Deployment Architecture . . . 53

13 Performance Optimization **54**
13.1 Caching Strategy 54
13.2 Parallel Processing 55

14 Evaluation and Metrics **56**
14.1 Epistemic Integrity Metrics 56
14.2 Performance Metrics 57

15 Testing Strategy **58**
15.1 Unit Testing 58
15.2 Integration Testing 59

16 Security and Privacy **60**
16.1 Data Privacy 60

17 Future Extensions **61**
17.1 Planned Enhancements 61

18 Conclusion **61**

A Appendix A: Database Schemas **62**

B Appendix B: Configuration Files **64**

C Appendix C: API Reference **65**

1 Executive Summary

1.1 Vision and Scope

KNOWY represents a paradigm shift from retrieval-oriented personal AI tools toward epistemic infrastructures. The system treats personal knowledge collections—spanning academic papers, courses, multimedia lectures, and saved references—not as static information repositories but as evolving epistemic foundations requiring rigorous integrity guarantees.

1.2 Core Design Principles

Foundational Design Principles

1. **Epistemic Integrity Over Information Access**: Prioritize correctness, uncertainty quantification, and hallucination prevention over comprehensive retrieval.
2. **Shallow-Deep Agent Separation**: Enforce strict boundaries between operational agents (bounded transformations) and epistemic agents (boundary-crossing reasoning).
3. **Multi-Resolution Knowledge Representation**: Maintain explicit abstraction hierarchies from raw artifacts to cross-document synthesis.
4. **Controlled External Research**: Enable knowledge evolution through principled, versioned external inquiry rather than unconstrained web access.
5. **Local-First Operation**: Ensure privacy, reproducibility, and system composability through command-line interfaces and local model execution.
6. **Multimodal Native Design**: Treat video, diagrams, and text as first-class knowledge modalities with specialized processing pipelines.

1.3 Key Innovations

- **Layered Epistemic Architecture**: Seven-layer shallow agent pipeline with isolated deep research component
- **Video-as-Knowledge Pipeline**: Integration of video-analyzer for shot segmentation, transcript alignment, and visual-semantic extraction
- **Diagrammatic Explanation Generation**: Automated schematic diagram creation for pedagogical enhancement
- **CLI-First Interaction Model**: Built on llm framework enabling scriptability and workflow integration
- **Versioned Knowledge Evolution**: Temporal knowledge management with explicit uncertainty tracking

2 System Architecture

2.1 Architectural Overview

The KNOWY architecture consists of eight distinct processing layers orchestrated through a graph-based execution framework. Seven layers implement shallow agents with bounded epistemic scope, while a single deep agent layer enables controlled external research.

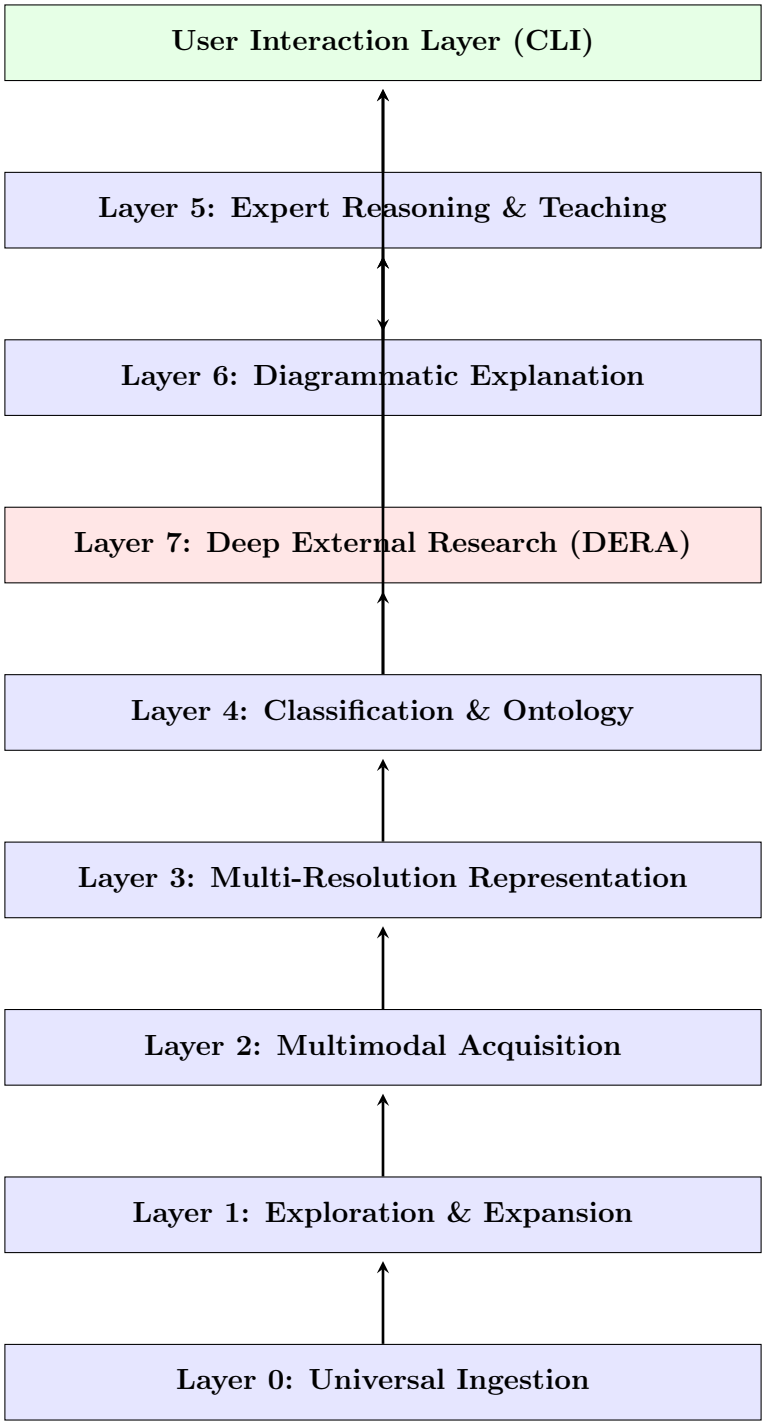

Figure 1: KNOWY Layered Architecture with Shallow-Deep Agent Separation

2.2 Architectural Guarantees

Guarantee	Implementation Mechanism
Epistemic Boundary Preservation	Type-level enforcement in orchestration graph; shallow agents cannot invoke external knowledge sources
Hallucination Minimization	Multi-resolution grounding with explicit provenance tracking; uncertainty quantification at reasoning layer
Temporal Consistency	Versioned knowledge representations with timestamp-based validity windows
Multimodal Coherence	Unified semantic representation across text, video frames, and diagrams with cross-modal alignment
Reproducibility	Deterministic agent execution with comprehensive logging; CLI-based workflow capture

Table 1: Core Architectural Guarantees

2.3 Technology Stack

Component	Technology	Justification
Agent Orchestration	LangGraph	Graph-based execution with conditional routing and state management
Local LLM Runtime	Ollama	Unified API for multiple local models; efficient inference
Text Models	LLaMA 3.x, Qwen, Gemma	State-of-the-art reasoning with local deployment
Multimodal Models	LLaMA 3.2 Vision	Native image understanding for video frame analysis
Video Analysis	video-analyzer	Shot segmentation, transcript alignment, visual-semantic extraction
CLI Framework	llm (Simon Willison)	Plugin architecture, scriptability, model abstraction
Vector Database	FAISS / Qdrant	High-performance similarity search with filtering
Structured Storage	PostgreSQL + Neo4j	Relational metadata + graph-based ontologies
Diagram Generation	Graphviz, Mermaid	Programmatic schematic creation

Table 2: Technology Stack Components

3 Layer 0: Universal Ingestion

3.1 Purpose and Epistemic Question

Epistemic Question: *"What reference has the user saved, independent of format or source?"*

Layer 0 establishes the entry point for all knowledge artifacts entering the system. It normalizes heterogeneous inputs from browsers, LinkedIn saves, manual entries, and API integrations into a canonical source descriptor format.

3.2 Canonical Source Descriptor Schema

Listing 1: Canonical Source Descriptor Data Structure

```python
from dataclasses import dataclass
from typing import List, Optional, Dict
from datetime import datetime
from uuid import UUID

@dataclass
class ContentHints:
    contains_pdf: bool
    contains_video: bool
    contains_external_links: bool
    contains_code_repository: bool
    estimated_length_minutes: Optional[int]
    primary_language: Optional[str]

@dataclass
class CanonicalSourceDescriptor:
    source_id: UUID
    origin: str  # linkedin | browser | manual | api | other
    raw_reference: str
    raw_text: Optional[str]
    links: List[str]
    content_hints: ContentHints
    timestamp: datetime
    ingestion_metadata: Dict[str, any]

    def to_json(self) -> dict:
        """Serialize to storage format"""
        pass

    @classmethod
    def from_raw_input(cls, raw_input: str, origin: str):
        """Factory method for ingestion"""
        pass
```

3.3 Ingestion Pipeline Architecture

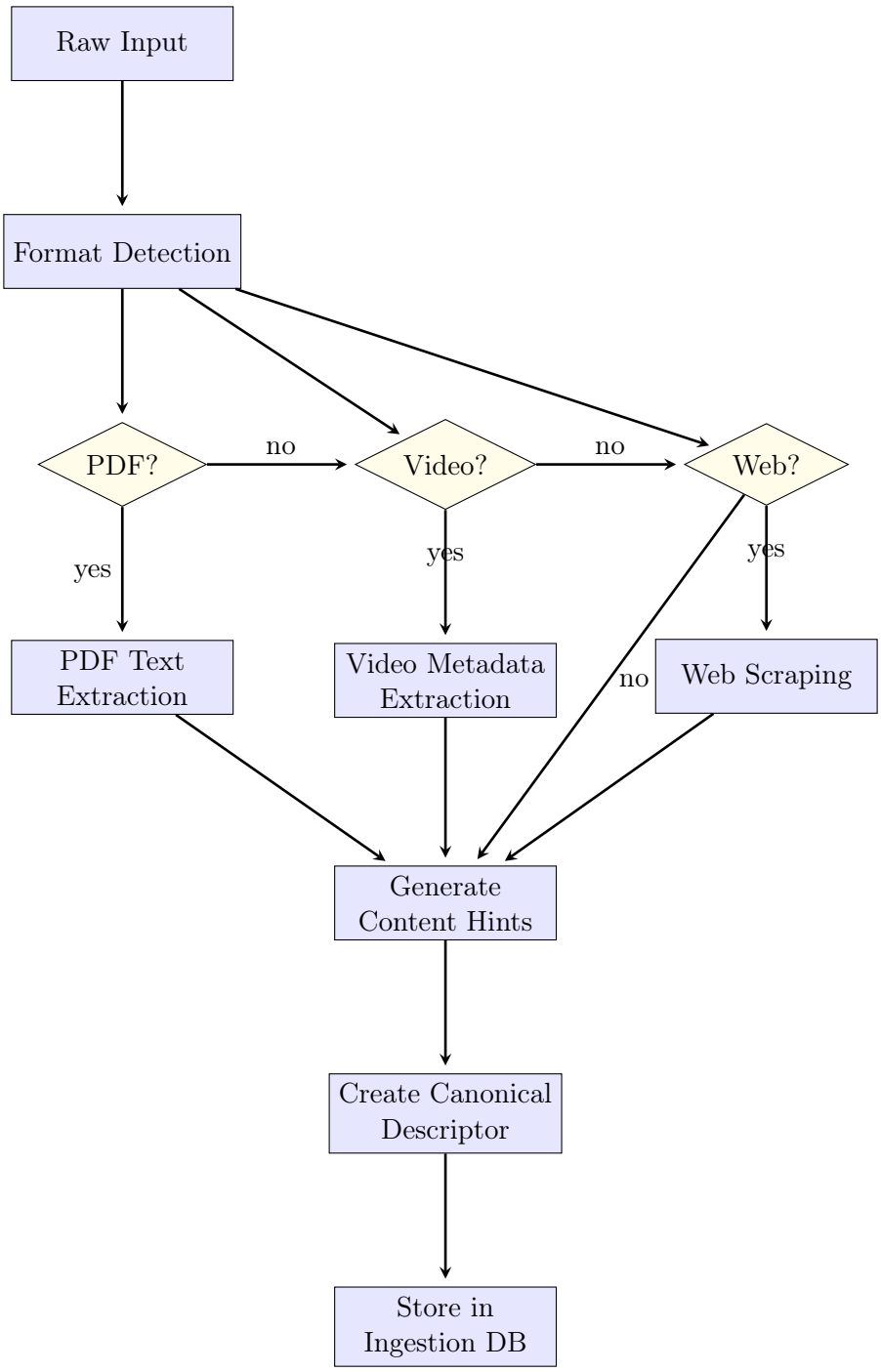

Figure 2: Layer 0 Ingestion Pipeline

3.4 Browser Extension Integration

Listing 2: Browser Extension Content Script

```
// KNOWY Browser Extension - Content Script
class KnowyExtension {
  constructor(apiEndpoint) {
    this.apiEndpoint = apiEndpoint;
  }
```

```
6
7   async captureCurrentPage() {
8     const pageData = {
9       url: window.location.href,
10      title: document.title,
11      selectedText: window.getSelection().toString(),
12      fullText: document.body.innerText,
13      links: Array.from(document.querySelectorAll('a'))
14        .map(a => a.href),
15      timestamp: new Date().toISOString()
16    };
17
18    return await this.sendToKnowy(pageData);
19  }
20
21  async sendToKnowy(data) {
22    const response = await fetch(`${this.apiEndpoint}/ingest`, {
23      method: 'POST',
24      headers: {'Content-Type': 'application/json'},
25      body: JSON.stringify({
26        origin: 'browser',
27        raw_reference: data.url,
28        raw_text: data.fullText,
29        links: data.links,
30        metadata: {title: data.title, selectedText: data.selectedText}
31      })
32    });
33    return response.json();
34  }
35 }
```

3.5 Format Detection Algorithm

Algorithm 1 Format Detection and Content Hint Generation

Require: Raw input string I, origin metadata O
Ensure: ContentHints object H

$H \leftarrow$ new ContentHints()
if I matches URL pattern **then**
  $domain \leftarrow$ extractDomain(I)
  if $domain \in$ {youtube.com, vimeo.com, coursera.org} **then**
    H.contains_video $\leftarrow$ true
    H.estimated_length_minutes $\leftarrow$ estimateFromMetadata(I)
  else if *domain* ends with .pdf **then**
    H.contains_pdf $\leftarrow$ true
  else
    $content \leftarrow$ fetchHTMLHead(I)
    $H \leftarrow$ analyzeOpenGraph($content$)
  end if
else if I matches file path pattern **then**
  $extension \leftarrow$ getExtension(I)
  $H \leftarrow$ mapExtensionToHints($extension$)
else
  H.contains_external_links $\leftarrow$ detectURLs(I)
end if
return H

3.6 Storage Schema

Listing 3: Ingestion Database Schema

```sql
CREATE TABLE ingestion_sources (
    source_id UUID PRIMARY KEY,
    origin VARCHAR(50) NOT NULL,
    raw_reference TEXT NOT NULL,
    raw_text TEXT,
    links JSONB,
    content_hints JSONB NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL,
    ingestion_metadata JSONB,
    processing_status VARCHAR(20) DEFAULT 'pending',
    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_origin ON ingestion_sources(origin);
CREATE INDEX idx_timestamp ON ingestion_sources(timestamp);
CREATE INDEX idx_status ON ingestion_sources(processing_status);
CREATE INDEX idx_hints_gin ON ingestion_sources USING GIN(content_hints
    );
```

4 Layer 1: Exploration and Expansion

4.1 Purpose and Epistemic Question

Epistemic Question: *"What actual knowledge artifacts exist behind this reference?"*

Layer 1 performs deep link analysis and reference expansion. A single user-saved URL may reference multiple papers, course modules, or video playlists that constitute distinct knowledge artifacts requiring separate processing.

4.2 Expansion Agent Architecture

Listing 4: Exploration Agent Implementation

```python
from typing import List, Set
from langchain.agents import AgentExecutor
from langchain.tools import Tool

class ExplorationAgent:
    def __init__(self, llm, web_browser_tool):
        self.llm = llm
        self.browser = web_browser_tool
        self.discovered_artifacts = []

    async def expand_reference(self,
                               source_descriptor:
                                   CanonicalSourceDescriptor
                              ) -> List[KnowledgeArtifact]:
        """
␣␣␣␣␣␣␣␣Expand␣a␣source␣descriptor␣into␣constituent␣knowledge␣artifacts
    .
␣␣␣␣␣␣␣␣"""
        artifacts = []

        # Analyze content hints to determine expansion strategy
        if source_descriptor.content_hints.contains_video:
            artifacts.extend(
                await self._expand_video_playlist(source_descriptor)
            )

        if source_descriptor.content_hints.contains_pdf:
            artifacts.extend(
                await self._expand_pdf_citations(source_descriptor)
            )

        if source_descriptor.content_hints.contains_external_links:
            artifacts.extend(
                await self._expand_linked_resources(source_descriptor)
            )

        # Use LLM to identify implicit references
        implicit_refs = await self._identify_implicit_references(
            source_descriptor.raw_text
        )
        artifacts.extend(implicit_refs)

        return self._deduplicate(artifacts)

    async def _expand_video_playlist(self,
                                     source: CanonicalSourceDescriptor
                                    ) -> List[KnowledgeArtifact]:
        """Extract␣individual␣videos␣from␣playlists␣or␣courses."""
        # Implementation uses platform-specific APIs
        pass
```

```python

    async def _expand_pdf_citations(self,
                                    source: CanonicalSourceDescriptor
                                   ) -> List[KnowledgeArtifact]:
        """Extract␣and␣resolve␣cited␣papers."""
        # Parse bibliography, resolve DOIs
        pass

    async def _identify_implicit_references(self,
                                            text: str
                                           ) -> List[KnowledgeArtifact
                                              ]:
        """Use␣LLM␣to␣identify␣mentioned␣resources."""
        prompt = f"""
␣␣␣␣␣␣␣␣Analyze␣the␣following␣text␣and␣identify␣any␣academic␣papers,
␣␣␣␣␣␣␣␣books,␣courses,␣or␣other␣knowledge␣resources␣that␣are␣mentioned
␣␣␣␣␣␣␣␣but␣not␣explicitly␣linked.␣For␣each,␣provide:
␣␣␣␣␣␣␣␣-␣Resource␣type␣(paper/book/course/video)
␣␣␣␣␣␣␣␣-␣Title
␣␣␣␣␣␣␣␣-␣Author(s)␣if␣mentioned
␣␣␣␣␣␣␣␣-␣Any␣identifying␣information␣(DOI,␣ISBN,␣URL)

␣␣␣␣␣␣␣␣Text:␣{text}

␣␣␣␣␣␣␣␣Return␣as␣structured␣JSON.
␣␣␣␣␣␣␣␣"""
        response = await self.llm.ainvoke(prompt)
        return self._parse_artifact_json(response)
```

4.3 Knowledge Artifact Data Structure

Listing 5: Knowledge Artifact Schema

```python
from enum import Enum

class ArtifactType(Enum):
    PAPER = "academic_paper"
    BOOK = "book"
    COURSE = "online_course"
    VIDEO_LECTURE = "video_lecture"
    BLOG_POST = "blog_post"
    DOCUMENTATION = "technical_documentation"
    DATASET = "dataset"
    CODE_REPOSITORY = "code_repository"

@dataclass
class KnowledgeArtifact:
    artifact_id: UUID
    source_id: UUID  # Links back to ingestion source
    artifact_type: ArtifactType
    title: str
    authors: List[str]
    url: Optional[str]
    doi: Optional[str]
    isbn: Optional[str]
    abstract: Optional[str]
    publication_date: Optional[datetime]
```

```
    acquisition_strategy: str  # Determines Layer 2 pipeline
    metadata: Dict[str, any]

    def requires_video_analysis(self) -> bool:
        return self.artifact_type in [
            ArtifactType.VIDEO_LECTURE,
            ArtifactType.COURSE
        ]

    def requires_pdf_processing(self) -> bool:
        return self.artifact_type in [
            ArtifactType.PAPER,
            ArtifactType.BOOK
        ] or (self.url and self.url.endswith('.pdf'))
```

4.4 Reference Resolution Strategies

Reference Type	Resolution Strategy	Tools/APIs
DOI	Query CrossRef API	`requests`, CrossRef REST API
arXiv ID	Parse arXiv metadata	arXiv API
ISBN	Query OpenLibrary	OpenLibrary API
YouTube URL	Extract video/playlist ID	YouTube Data API
Course Platform	Platform-specific scraping	Selenium, BeautifulSoup
GitHub Repository	Clone and analyze structure	PyGithub, git
Generic Web Page	Content extraction	Trafilatura, newspaper3k

Table 3: Reference Resolution Strategies by Type

4.5 Deduplication Algorithm

Algorithm 2 Knowledge Artifact Deduplication

Require: Set of artifacts $A = \{a_1, a_2, \ldots, a_n\}$
Ensure: Deduplicated set A'

$A' \leftarrow \emptyset$
$seen \leftarrow \emptyset$
for $a \in A$ **do**
  $key \leftarrow \text{generateKey}(a)$
  if $key \notin seen$ **then**
    $A' \leftarrow A' \cup \{a\}$
    $seen \leftarrow seen \cup \{key\}$
  else
    $existing \leftarrow \text{findByKey}(A', key)$
    $merged \leftarrow \text{mergeMetadata}(existing, a)$
    $\text{replace}(existing, merged)$
  end if
end for
return A'

Where the key generation function prioritizes unique identifiers:

Listing 6: Artifact Key Generation

```python
def generate_key(artifact: KnowledgeArtifact) -> str:
    if artifact.doi:
        return f"doi:{artifact.doi}"
    elif artifact.isbn:
        return f"isbn:{artifact.isbn}"
    elif artifact.url:
        return f"url:{normalize_url(artifact.url)}"
    else:
        # Fallback to content-based hash
        return f"hash:{hash_content(artifact.title, artifact.authors)}"
```

5 Layer 2: Multimodal Acquisition and Scraping

5.1 Purpose and Epistemic Question

Epistemic Question: *"What is the complete primary content of each identified artifact?"*

Layer 2 implements specialized acquisition pipelines for different modalities and formats. The most sophisticated component is the video analysis pipeline, which transforms long-form lectures into structured knowledge units.

5.2 Acquisition Pipeline Dispatcher

Listing 7: Acquisition Pipeline Orchestration

```python
class AcquisitionOrchestrator:
    def __init__(self):
        self.pipelines = {
            'pdf': PDFAcquisitionPipeline(),
            'video': VideoAnalysisPipeline(),
            'web': WebScrapingPipeline(),
            'code': CodeRepositoryPipeline(),
            'paper': AcademicPaperPipeline()
        }

    async def acquire(self,
                      artifact: KnowledgeArtifact
                     ) -> AcquiredContent:
        pipeline = self._select_pipeline(artifact)

        try:
            content = await pipeline.acquire(artifact)
            validation_result = await self._validate_content(content)

            if validation_result.is_valid:
                return content
            else:
                # Attempt fallback pipeline
                fallback = self._get_fallback_pipeline(artifact)
                return await fallback.acquire(artifact)

        except AcquisitionException as e:
            # Log failure and return partial content
            self._log_acquisition_failure(artifact, e)
```

```
            return self._create_partial_content(artifact, e)

    def _select_pipeline(self,
                         artifact: KnowledgeArtifact
                        ) -> AcquisitionPipeline:
        if artifact.requires_video_analysis():
            return self.pipelines['video']
        elif artifact.requires_pdf_processing():
            if artifact.doi or artifact.artifact_type == ArtifactType.
               PAPER:
                return self.pipelines['paper']
            else:
                return self.pipelines['pdf']
        elif artifact.artifact_type == ArtifactType.CODE_REPOSITORY:
            return self.pipelines['code']
        else:
            return self.pipelines['web']
```

5.3 Video Analysis Pipeline Integration

The video analysis pipeline integrates the open-source `video-analyzer` project to perform shot-level semantic extraction.

5.3.1 Video Analysis Architecture

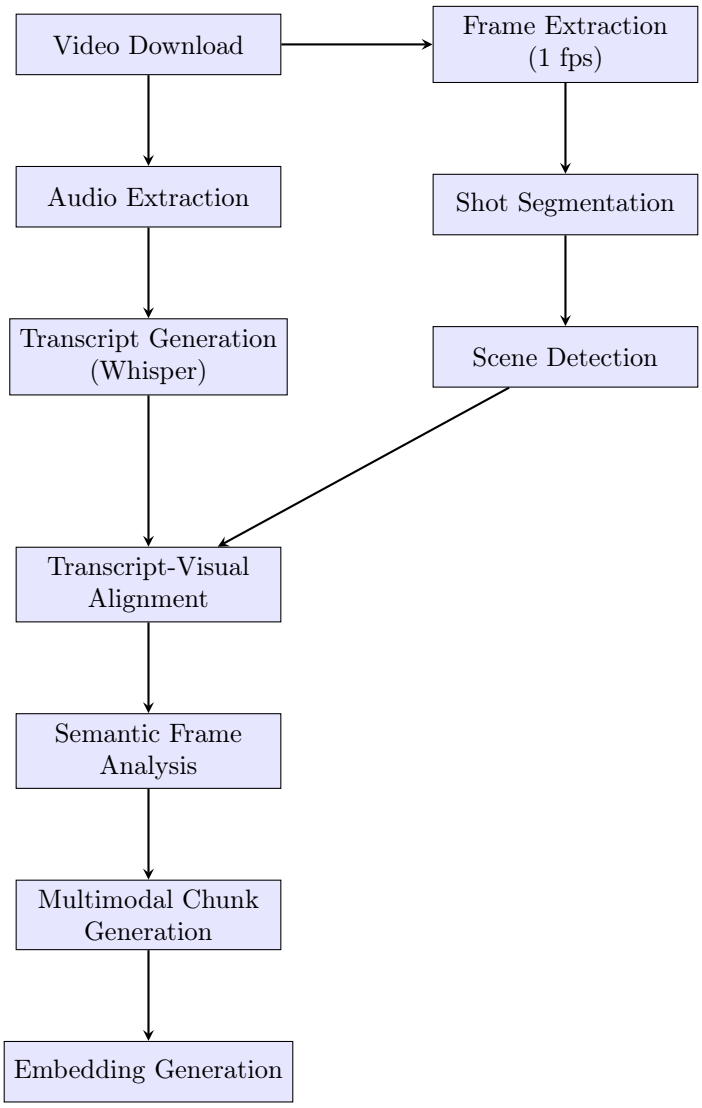

Figure 3: Video Analysis Pipeline Architecture

5.3.2 Shot Segmentation Algorithm

Algorithm 3 Adaptive Shot Boundary Detection

Require: Video frames $F = \{f_1, f_2, \ldots, f_n\}$, threshold τ
Ensure: Shot boundaries $B = \{b_1, b_2, \ldots, b_m\}$

$B \leftarrow \emptyset$
$prev\_hist \leftarrow \text{computeHistogram}(f_1)$
for $i \leftarrow 2$ to n **do**
  $curr\_hist \leftarrow \text{computeHistogram}(f_i)$
  $diff \leftarrow \text{histogramDistance}(prev\_hist, curr\_hist)$
  if $diff > \tau$ **then**
    $B \leftarrow B \cup \{i\}$
    $\tau \leftarrow \text{adaptThreshold}(\tau, diff)$
  end if
  $prev\_hist \leftarrow curr\_hist$
end for
return B

5.3.3 Transcript-Visual Alignment

Listing 8: Transcript-Visual Temporal Alignment

```python
class TranscriptVisualAligner:
    def __init__(self, whisper_model):
        self.whisper = whisper_model

    async def align(self,
                    transcript_segments: List[TranscriptSegment],
                    shots: List[Shot]) -> List[AlignedChunk]:
        """
␣␣␣␣␣␣␣␣Align␣transcript␣segments␣with␣visual␣shots␣to␣create
␣␣␣␣␣␣␣␣semantically␣coherent␣multimodal␣chunks.
␣␣␣␣␣␣␣␣"""
        aligned_chunks = []
        shot_idx = 0

        for segment in transcript_segments:
            # Find overlapping shots
            overlapping_shots = []
            while (shot_idx < len(shots) and
                   shots[shot_idx].start_time <= segment.end_time):
                if shots[shot_idx].end_time >= segment.start_time:
                    overlapping_shots.append(shots[shot_idx])
                shot_idx += 1

            # Create aligned chunk
            chunk = AlignedChunk(
                start_time=segment.start_time,
                end_time=segment.end_time,
                transcript_text=segment.text,
                shots=overlapping_shots,
                speaker=segment.speaker,
                confidence=segment.confidence
            )

```

```
            aligned_chunks.append(chunk)

        return aligned_chunks

@dataclass
class AlignedChunk:
    start_time: float
    end_time: float
    transcript_text: str
    shots: List[Shot]
    speaker: Optional[str]
    confidence: float

    def get_representative_frame(self) -> np.ndarray:
        """Select most informative frame from shots."""
        if not self.shots:
            return None
        # Use middle frame of longest shot
        longest_shot = max(self.shots, key=lambda s: s.duration)
        return longest_shot.frames[len(longest_shot.frames) // 2]
```

5.3.4 Semantic Frame Analysis

Listing 9: Visual-Semantic Frame Analysis with LLaMA Vision

```
class SemanticFrameAnalyzer:
    def __init__(self, vision_model):
        self.vision_model = vision_model  # LLaMA 3.2 Vision

    async def analyze_chunk(self,
                            chunk: AlignedChunk
                           ) -> SemanticAnalysis:
        """
        Perform semantic analysis of visual content in context
        of transcript.
        """
        frame = chunk.get_representative_frame()

        prompt = f"""
        Analyze this frame from an educational video lecture.

        Transcript context: "{chunk.transcript_text}"

        Describe:
        1. Visual content (diagrams, equations, code, slides)
        2. Key concepts being illustrated
        3. Relationship between visual and spoken content
        4. Educational value of this visual

        Provide structured analysis.
        """

        analysis = await self.vision_model.analyze(
            image=frame,
            prompt=prompt
        )

```

```
        return SemanticAnalysis(
            chunk_id=chunk.id,
            visual_description=analysis.visual_content,
            concepts=analysis.key_concepts,
            visual_transcript_coherence=analysis.coherence_score,
            educational_value=analysis.educational_value,
            contains_diagram=self._detect_diagram(analysis),
            contains_equation=self._detect_equation(analysis),
            contains_code=self._detect_code(analysis)
        )
```

5.3.5 Multimodal Chunk Generation

Listing 10: Complete Multimodal Chunk Data Structure

```
@dataclass
class MultimodalChunk:
    chunk_id: UUID
    artifact_id: UUID
    temporal_range: Tuple[float, float]

    # Textual components
    transcript_text: str
    transcript_confidence: float

    # Visual components
    representative_frame: bytes  # JPEG-encoded
    all_frames: List[bytes]
    shot_boundaries: List[float]

    # Semantic analysis
    visual_description: str
    key_concepts: List[str]
    contains_diagram: bool
    contains_equation: bool
    contains_code: bool

    # Embeddings
    text_embedding: np.ndarray
    visual_embedding: np.ndarray
    combined_embedding: np.ndarray

    # Metadata
    speaker: Optional[str]
    slide_number: Optional[int]
    topic_tags: List[str]

    def to_storage_format(self) -> dict:
        """Prepare for database insertion."""
        return {
            'chunk_id': str(self.chunk_id),
            'artifact_id': str(self.artifact_id),
            'temporal_range': self.temporal_range,
            'transcript_text': self.transcript_text,
            'visual_description': self.visual_description,
            'key_concepts': json.dumps(self.key_concepts),
            'semantic_flags': {
```

```
                'diagram': self.contains_diagram,
                'equation': self.contains_equation,
                'code': self.contains_code
            },
            'embeddings': {
                'text': self.text_embedding.tolist(),
                'visual': self.visual_embedding.tolist(),
                'combined': self.combined_embedding.tolist()
            }
        }
```

5.4 PDF Acquisition Pipeline

Listing 11: PDF Processing with Layout Preservation

```
class PDFAcquisitionPipeline:
    def __init__(self):
        self.extractors = {
            'text': PDFTextExtractor(),
            'layout': PDFLayoutExtractor(),
            'images': PDFImageExtractor(),
            'tables': PDFTableExtractor()
        }

    async def acquire(self, artifact: KnowledgeArtifact) -> PDFContent:
        pdf_bytes = await self._download_pdf(artifact.url)

        # Parallel extraction
        text_task = self.extractors['text'].extract(pdf_bytes)
        layout_task = self.extractors['layout'].extract(pdf_bytes)
        images_task = self.extractors['images'].extract(pdf_bytes)
        tables_task = self.extractors['tables'].extract(pdf_bytes)

        text, layout, images, tables = await asyncio.gather(
            text_task, layout_task, images_task, tables_task
        )

        # Reconstruct document structure
        structured_content = self._reconstruct_structure(
            text, layout, images, tables
        )

        return PDFContent(
            artifact_id=artifact.artifact_id,
            raw_text=text.full_text,
            sections=structured_content.sections,
            figures=images,
            tables=tables,
            citations=self._extract_citations(text),
            metadata=self._extract_metadata(pdf_bytes)
        )

    def _reconstruct_structure(self, text, layout, images, tables):
        """
        Reconstruct document hierarchy using layout analysis.
        Preserves section headers, paragraphs, lists, etc.
        """
```

```
        sections = []
        current_section = None

        for block in layout.blocks:
            if block.type == 'heading':
                if current_section:
                    sections.append(current_section)
                current_section = Section(
                    title=block.text,
                    level=block.heading_level,
                    content=[]
                )
            elif block.type == 'paragraph':
                if current_section:
                    current_section.content.append(
                        Paragraph(text=block.text)
                    )
            elif block.type == 'figure':
                fig = self._match_figure(block, images)
                if current_section and fig:
                    current_section.content.append(fig)
            elif block.type == 'table':
                tbl = self._match_table(block, tables)
                if current_section and tbl:
                    current_section.content.append(tbl)

        if current_section:
            sections.append(current_section)

        return StructuredDocument(sections=sections)
```

5.5 Academic Paper Pipeline

Listing 12: Academic Paper Specialized Processing

```
class AcademicPaperPipeline:
    def __init__(self):
        self.pdf_pipeline = PDFAcquisitionPipeline()
        self.metadata_enricher = PaperMetadataEnricher()

    async def acquire(self, artifact: KnowledgeArtifact) ->
        PaperContent:
        # First get PDF content
        pdf_content = await self.pdf_pipeline.acquire(artifact)

        # Enrich with academic metadata
        metadata = await self.metadata_enricher.enrich(
            doi=artifact.doi,
            arxiv_id=self._extract_arxiv_id(artifact.url)
        )

        # Parse academic structure
        academic_structure = self._parse_academic_structure(
            pdf_content
        )

        return PaperContent(
```

```
            artifact_id=artifact.artifact_id,
            title=metadata.title or artifact.title,
            authors=metadata.authors or artifact.authors,
            abstract=academic_structure.abstract,
            introduction=academic_structure.introduction,
            methodology=academic_structure.methodology,
            results=academic_structure.results,
            discussion=academic_structure.discussion,
            conclusion=academic_structure.conclusion,
            references=pdf_content.citations,
            figures=pdf_content.figures,
            tables=pdf_content.tables,
            equations=self._extract_equations(pdf_content),
            citation_count=metadata.citation_count,
            publication_venue=metadata.venue,
            publication_date=metadata.publication_date
        )

    def _parse_academic_structure(self, pdf_content):
        """
␣␣␣␣␣␣␣␣Use␣section␣headers␣to␣identify␣standard␣paper␣sections.
␣␣␣␣␣␣␣␣"""
        section_map = {
            'abstract': None,
            'introduction': None,
            'methodology': None,
            'results': None,
            'discussion': None,
            'conclusion': None
        }

        for section in pdf_content.sections:
            title_lower = section.title.lower()

            if 'abstract' in title_lower:
                section_map['abstract'] = section
            elif 'introduction' in title_lower:
                section_map['introduction'] = section
            elif any(kw in title_lower for kw in
                    ['method', 'approach', 'design']):
                section_map['methodology'] = section
            elif 'result' in title_lower:
                section_map['results'] = section
            elif 'discussion' in title_lower:
                section_map['discussion'] = section
            elif 'conclusion' in title_lower:
                section_map['conclusion'] = section

        return AcademicStructure(**section_map)
```

6 Layer 3: Multi-Resolution Knowledge Representation

6.1 Purpose and Epistemic Question

Epistemic Question: *"How should knowledge be structured at multiple levels of abstraction to support both factual grounding and high-level reasoning?"*

Layer 3 constructs a four-level abstraction hierarchy that enables both precise fact retrieval and synthesized understanding.

6.2 Four-Level Abstraction Hierarchy

Level	Granularity	Use Case
Level 0	Raw chunks (paragraphs, video segments)	Direct citation, fact verification
Level 1	Section-level summaries	Navigating document structure
Level 2	Document-level synthesis	Comparing papers, course overview
Level 3	Cross-document thematic synthesis	Domain understanding, research frontiers

Table 4: Multi-Resolution Abstraction Levels

6.3 Representation Architecture

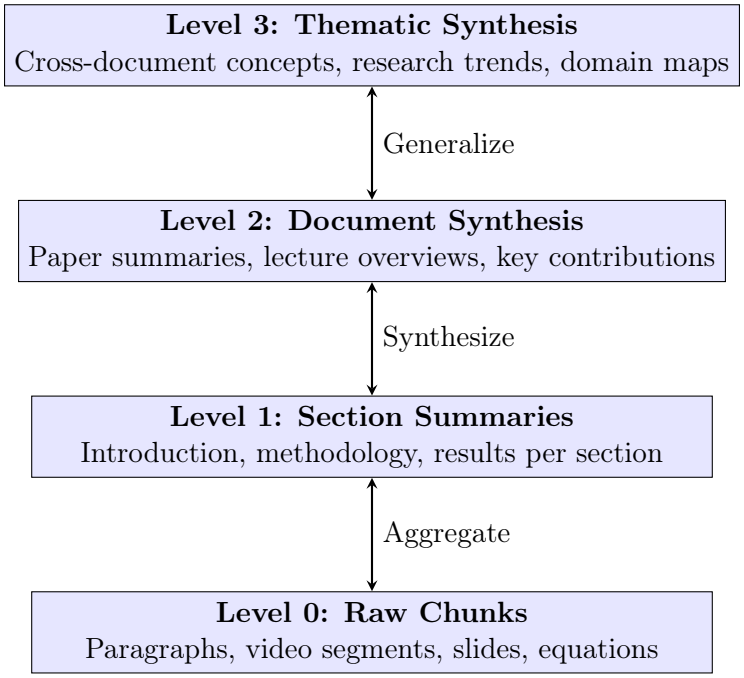

Figure 4: Multi-Resolution Representation Hierarchy

6.4 Level 0: Raw Content Storage

Listing 13: Level 0 Raw Chunk Representation

```
@dataclass
class RawChunk:
    chunk_id: UUID
    artifact_id: UUID
    chunk_type: str  # paragraph | video_segment | equation | figure
```

```
    # Content
    text_content: Optional[str]
    visual_content: Optional[bytes]
    structured_content: Optional[dict]  # For tables, code

    # Positional metadata
    position_in_artifact: int
    section_id: Optional[UUID]
    temporal_range: Optional[Tuple[float, float]]  # For video
    page_number: Optional[int]  # For PDFs

    # Embeddings
    embedding: np.ndarray

    # Provenance
    extraction_method: str
    confidence: float

    def get_context_window(self,
                           window_size: int = 2
                          ) -> List['RawChunk']:
        """Retrieve neighboring chunks for context."""
        pass

# Storage schema
CREATE TABLE raw_chunks (
    chunk_id UUID PRIMARY KEY,
    artifact_id UUID REFERENCES knowledge_artifacts(artifact_id),
    chunk_type VARCHAR(50),
    text_content TEXT,
    visual_content BYTEA,
    structured_content JSONB,
    position_in_artifact INTEGER,
    section_id UUID,
    temporal_range NUMRANGE,
    page_number INTEGER,
    embedding vector(1536),
    extraction_metadata JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_artifact ON raw_chunks(artifact_id);
CREATE INDEX idx_section ON raw_chunks(section_id);
CREATE INDEX idx_embedding ON raw_chunks
    USING ivfflat (embedding vector_cosine_ops);
```

6.5 Level 1: Section-Level Representation

Listing 14: Level 1 Section Summary Generation

```
class SectionSummarizer:
    def __init__(self, llm):
        self.llm = llm

    async def generate_section_summary(self,
                                       section_chunks: List[RawChunk]
```

```
                                        ) -> SectionRepresentation:
        """
␣␣␣␣␣␣␣␣Generate␣structured␣summary␣of␣a␣document␣section.
␣␣␣␣␣␣␣␣"""
        # Concatenate section content
        full_text = "
".join([
            chunk.text_content for chunk in section_chunks
            if chunk.text_content
        ])

        prompt = f"""
␣␣␣␣␣␣␣␣Summarize␣the␣following␣section␣from␣an␣academic␣document.
␣␣␣␣␣␣␣␣Provide:
␣␣␣␣␣␣␣␣1.␣Main␣claim␣or␣thesis␣(1-2␣sentences)
␣␣␣␣␣␣␣␣2.␣Key␣points␣(bullet␣list)
␣␣␣␣␣␣␣␣3.␣Important␣terminology␣defined
␣␣␣␣␣␣␣␣4.␣Figures/equations␣referenced
␣␣␣␣␣␣␣␣5.␣Dependencies␣on␣other␣sections

␣␣␣␣␣␣␣␣Section␣content:
␣␣␣␣␣␣␣␣{full_text}

␣␣␣␣␣␣␣␣Output␣as␣structured␣JSON.
␣␣␣␣␣␣␣␣"""

        summary = await self.llm.ainvoke(prompt)
        parsed = json.loads(summary)

        return SectionRepresentation(
            section_id=uuid4(),
            artifact_id=section_chunks[0].artifact_id,
            title=self._extract_section_title(section_chunks),
            main_claim=parsed['main_claim'],
            key_points=parsed['key_points'],
            terminology=parsed['terminology'],
            referenced_elements=parsed['references'],
            chunk_ids=[c.chunk_id for c in section_chunks],
            embedding=await self._embed_summary(parsed['main_claim'])
        )

@dataclass
class SectionRepresentation:
    section_id: UUID
    artifact_id: UUID
    title: str
    main_claim: str
    key_points: List[str]
    terminology: Dict[str, str]
    referenced_elements: List[str]
    chunk_ids: List[UUID]
    embedding: np.ndarray
    prerequisites: List[UUID] = field(default_factory=list)
```

6.6 Level 2: Document-Level Synthesis

Listing 15: Level 2 Document Synthesis

```
class DocumentSynthesizer:
    def __init__(self, llm):
        self.llm = llm

    async def synthesize_document(self,
                                  sections: List[SectionRepresentation]
                                 ) -> DocumentRepresentation:
        """
␣␣␣␣␣␣␣␣Create␣document-level␣synthesis␣from␣section␣summaries.
␣␣␣␣␣␣␣␣"""
        # Construct synthesis prompt
        section_summaries = "
".join([
            f"Section:␣{s.title}
Main␣claim:␣{s.main_claim}
"
            for s in sections
        ])

        prompt = f"""
␣␣␣␣␣␣␣␣Synthesize␣the␣following␣academic␣document␣into␣a␣cohesive␣
    summary.

␣␣␣␣␣␣␣␣Sections:
␣␣␣␣␣␣␣␣{section_summaries}

␣␣␣␣␣␣␣␣Provide:
␣␣␣␣␣␣␣␣1.␣Central␣thesis␣(2-3␣sentences)
␣␣␣␣␣␣␣␣2.␣Main␣contributions␣(numbered␣list)
␣␣␣␣␣␣␣␣3.␣Methodology␣overview
␣␣␣␣␣␣␣␣4.␣Key␣findings
␣␣␣␣␣␣␣␣5.␣Limitations␣discussed
␣␣␣␣␣␣␣␣6.␣Related␣work␣positioning
␣␣␣␣␣␣␣␣7.␣Novel␣concepts␣introduced

␣␣␣␣␣␣␣␣Output␣as␣structured␣JSON.
␣␣␣␣␣␣␣␣"""

        synthesis = await self.llm.ainvoke(prompt)
        parsed = json.loads(synthesis)

        return DocumentRepresentation(
            document_id=uuid4(),
            artifact_id=sections[0].artifact_id,
            central_thesis=parsed['central_thesis'],
            contributions=parsed['contributions'],
            methodology=parsed['methodology'],
            findings=parsed['findings'],
            limitations=parsed['limitations'],
            novel_concepts=parsed['novel_concepts'],
            section_ids=[s.section_id for s in sections],
            embedding=await self._embed_synthesis(
                parsed['central_thesis']
            )
        )

@dataclass
```

```python
class DocumentRepresentation:
    document_id: UUID
    artifact_id: UUID
    central_thesis: str
    contributions: List[str]
    methodology: str
    findings: List[str]
    limitations: List[str]
    novel_concepts: List[str]
    section_ids: List[UUID]
    embedding: np.ndarray
    citation_graph: Optional[dict] = None
```

6.7 Level 3: Cross-Document Thematic Synthesis

Listing 16: Level 3 Thematic Synthesis

```python
class ThematicSynthesizer:
    def __init__(self, llm, clustering_model):
        self.llm = llm
        self.clusterer = clustering_model

    async def synthesize_themes(self,
                                documents: List[DocumentRepresentation]
                               ) -> List[ThematicSynthesis]:
        """
␣␣␣␣␣␣␣␣Identify␣cross-document␣themes␣and␣research␣trends.
␣␣␣␣␣␣␣␣"""
        # Cluster documents by embedding similarity
        embeddings = np.array([d.embedding for d in documents])
        clusters = self.clusterer.fit_predict(embeddings)

        themes = []
        for cluster_id in np.unique(clusters):
            cluster_docs = [
                documents[i] for i in range(len(documents))
                if clusters[i] == cluster_id
            ]

            theme = await self._synthesize_cluster(cluster_docs)
            themes.append(theme)

        return themes

    async def _synthesize_cluster(self,
                                  docs: List[DocumentRepresentation]
                                 ) -> ThematicSynthesis:
        """
␣␣␣␣␣␣␣␣Synthesize␣a␣thematic␣cluster␣into␣unified␣understanding.
␣␣␣␣␣␣␣␣"""
        # Extract key concepts across documents
        all_concepts = []
        all_theses = []

        for doc in docs:
            all_concepts.extend(doc.novel_concepts)
            all_theses.append(doc.central_thesis)
```

```
        prompt = f"""
␣␣␣␣␣␣␣␣Analyze␣the␣following␣related␣research␣documents␣and␣identify:

␣␣␣␣␣␣␣␣1.␣Common␣theme␣or␣research␣question
␣␣␣␣␣␣␣␣2.␣Points␣of␣agreement
␣␣␣␣␣␣␣␣3.␣Points␣of␣disagreement/debate
␣␣␣␣␣␣␣␣4.␣Evolution␣of␣ideas␣(temporal␣if␣dates␣available)
␣␣␣␣␣␣␣␣5.␣Open␣questions␣in␣this␣area
␣␣␣␣␣␣␣␣6.␣Key␣terminology␣and␣definitions

␣␣␣␣␣␣␣␣Document␣theses:
␣␣␣␣␣␣␣␣{chr(10).join([f"- {t}"␣for␣t␣in␣all_theses])}

␣␣␣␣␣␣␣␣Novel␣concepts␣mentioned:
␣␣␣␣␣␣␣␣{chr(10).join([f"- {c}"␣for␣c␣in␣set(all_concepts)])}

␣␣␣␣␣␣␣␣Output␣as␣structured␣JSON.
␣␣␣␣␣␣␣␣"""

        synthesis = await self.llm.ainvoke(prompt)
        parsed = json.loads(synthesis)

        return ThematicSynthesis(
            theme_id=uuid4(),
            theme_name=parsed['common_theme'],
            document_ids=[d.document_id for d in docs],
            agreements=parsed['agreements'],
            disagreements=parsed['disagreements'],
            evolution=parsed['evolution'],
            open_questions=parsed['open_questions'],
            key_terminology=parsed['terminology'],
            embedding=await self._embed_theme(parsed['common_theme'])
        )

@dataclass
class ThematicSynthesis:
    theme_id: UUID
    theme_name: str
    document_ids: List[UUID]
    agreements: List[str]
    disagreements: List[str]
    evolution: Optional[str]
    open_questions: List[str]
    key_terminology: Dict[str, str]
    embedding: np.ndarray
    research_frontier: bool = False
```

6.8 Dynamic Resolution Selection

Listing 17: Query-Adaptive Resolution Selection

```
class ResolutionSelector:
    def __init__(self):
        self.query_classifier = QueryTypeClassifier()

    def select_optimal_level(self, query: str) -> int:
```

```
        """
␣␣␣␣␣␣␣␣Determine␣optimal␣abstraction␣level␣for␣query.
␣␣␣␣␣␣␣␣"""
        query_type = self.query_classifier.classify(query)

        if query_type == 'fact_lookup':
            # Need precise citation
            return 0
        elif query_type == 'section_comparison':
            # Compare methodologies, results sections
            return 1
        elif query_type == 'paper_summary':
            # Overview of single document
            return 2
        elif query_type == 'domain_understanding':
            # Conceptual understanding across sources
            return 3
        elif query_type == 'pedagogical':
            # Teaching requires multiple levels
            return [0, 1, 2, 3]  # Multi-level retrieval
        else:
            # Default to document level
            return 2
```

7 Layer 4: Classification and Ontology Construction

7.1 Purpose and Epistemic Question

Epistemic Question: *"How are knowledge artifacts related topically, hierarchically, and temporally?"*

Layer 4 constructs explicit ontologies capturing topic hierarchies, prerequisite relationships, and redundancy mappings.

7.2 Ontology Structure

Listing 18: Knowledge Ontology Data Model

```
@dataclass
class TopicNode:
    topic_id: UUID
    name: str
    parent_topic: Optional[UUID]
    child_topics: List[UUID]
    related_documents: List[UUID]
    related_sections: List[UUID]

    # Topic characterization
    description: str
    key_terminology: List[str]
    difficulty_level: int  # 1-5

    # Prerequisites
    prerequisite_topics: List[UUID]
    recommended_learning_order: int

    def is_leaf(self) -> bool:
```

```
        return len(self.child_topics) == 0

    def get_ancestors(self) -> List['TopicNode']:
        """Traverse to root to get full path."""
        pass

@dataclass
class PrerequisiteRelationship:
    prerequisite_id: UUID
    dependent_id: UUID
    strength: float  # 0-1, how essential is prerequisite
    relationship_type: str  # foundational | helpful | optional

# Graph storage schema (Neo4j Cypher)
CREATE (t:Topic {
    topic_id: $topic_id,
    name: $name,
    description: $description,
    difficulty: $difficulty
})

CREATE (t1:Topic)-[:PARENT_OF]->(t2:Topic)
CREATE (t1:Topic)-[:PREREQUISITE_FOR {strength: 0.9}]->(t2:Topic)
CREATE (d:Document)-[:COVERS]->(t:Topic)
```

7.3 Topic Extraction and Clustering

Listing 19: Automated Topic Extraction

```
class TopicExtractor:
    def __init__(self, llm, embedding_model):
        self.llm = llm
        self.embedder = embedding_model

    async def extract_topics(self,
                             documents: List[DocumentRepresentation]
                            ) -> List[TopicNode]:
        """
        Extract hierarchical topic structure from documents.
        """
        # Step 1: Extract candidate topics from each document
        all_candidates = []
        for doc in documents:
            candidates = await self._extract_document_topics(doc)
            all_candidates.extend(candidates)

        # Step 2: Cluster and deduplicate topics
        unified_topics = await self._unify_topics(all_candidates)

        # Step 3: Construct hierarchy
        hierarchy = await self._build_hierarchy(unified_topics)

        # Step 4: Identify prerequisites
        await self._identify_prerequisites(hierarchy)

        return hierarchy

```

```
    async def _extract_document_topics(self,
                                       doc: DocumentRepresentation
                                      ) -> List[str]:
        """Extract topics mentioned in document."""
        prompt = f"""
        Extract the main topics and subtopics covered in this document.

        Thesis: {doc.central_thesis}
        Contributions: {doc.contributions}
        Novel concepts: {doc.novel_concepts}

        For each topic, provide:
        - Topic name
        - Brief description
        - Estimated difficulty level (1-5)

        Output as JSON list.
        """

        response = await self.llm.ainvoke(prompt)
        return json.loads(response)

    async def _unify_topics(self,
                            candidates: List[dict]
                           ) -> List[TopicNode]:
        """
        Cluster similar topics and create unified representations.
        """
        # Embed all topic names and descriptions
        embeddings = []
        for candidate in candidates:
            emb = await self.embedder.embed(
                f"{candidate['name']}: {candidate['description']}"
            )
            embeddings.append(emb)

        # Cluster by similarity
        clusterer = DBSCAN(eps=0.3, min_samples=2)
        clusters = clusterer.fit_predict(np.array(embeddings))

        unified = []
        for cluster_id in np.unique(clusters):
            cluster_topics = [
                candidates[i] for i in range(len(candidates))
                if clusters[i] == cluster_id
            ]

            # Merge cluster into single topic
            merged = await self._merge_topic_cluster(cluster_topics)
            unified.append(TopicNode(**merged))

        return unified

    async def _build_hierarchy(self,
                               topics: List[TopicNode]
                              ) -> List[TopicNode]:
        """
        Construct parent-child relationships between topics.
```

```
        """
        prompt = f"""
        Given these topics, construct a hierarchical taxonomy.
        Identify which topics are subtopics of others.

        Topics:
        {json.dumps([{'id': str(t.topic_id), 'name': t.name, 'description': t.description} for t in topics], indent=2)}

        Output parent-child relationships as JSON:
        [{{"parent_id": "...", "child_id": "..."}}, ...]
        """

        relationships = await self.llm.ainvoke(prompt)
        parsed = json.loads(relationships)

        # Apply relationships to topic nodes
        for rel in parsed:
            parent = next(t for t in topics if str(t.topic_id) == rel['parent_id'])
            child = next(t for t in topics if str(t.topic_id) == rel['child_id'])

            parent.child_topics.append(child.topic_id)
            child.parent_topic = parent.topic_id

        return topics

    async def _identify_prerequisites(self,
                                      topics: List[TopicNode]):
        """
        Identify prerequisite relationships between topics.
        """
        for topic in topics:
            prompt = f"""
            For the topic "{topic.name}": {topic.description}

            From this list of other topics, identify which are prerequisites:
            {json.dumps([{'id': str(t.topic_id), 'name': t.name} for t in topics if t.topic_id != topic.topic_id], indent=2)}

            For each prerequisite, rate its importance:
            - foundational (0.9-1.0): Cannot understand topic without this
            - helpful (0.5-0.8): Significantly aids understanding
            - optional (0.1-0.4): Provides context but not required

            Output as JSON: [{{"topic_id": "...", "strength": 0.9, "type": "foundational"}}, ...]
            """

            prereqs = await self.llm.ainvoke(prompt)
            parsed = json.loads(prereqs)

            for prereq in parsed:
                topic.prerequisite_topics.append(UUID(prereq['topic_id']))
```

7.4 Redundancy Detection

Listing 20: Content Redundancy Detection and Mapping

```python
class RedundancyDetector:
    def __init__(self, similarity_threshold=0.85):
        self.threshold = similarity_threshold

    def detect_redundancy(self,
                          sections: List[SectionRepresentation]
                         ) -> List[RedundancyGroup]:
        """
␣␣␣␣␣␣␣␣Identify␣sections␣covering␣similar␣content␣across␣documents.
␣␣␣␣␣␣␣␣"""
        embeddings = np.array([s.embedding for s in sections])

        # Compute pairwise cosine similarities
        similarities = cosine_similarity(embeddings)

        # Find redundant groups
        redundancy_groups = []
        processed = set()

        for i in range(len(sections)):
            if i in processed:
                continue

            # Find all sections similar to section i
            similar_indices = np.where(
                similarities[i] > self.threshold
            )[0]

            if len(similar_indices) > 1:
                group = RedundancyGroup(
                    group_id=uuid4(),
                    section_ids=[sections[idx].section_id
                                 for idx in similar_indices],
                    similarity_scores=similarities[i][similar_indices].
                       tolist(),
                    canonical_section=sections[i].section_id,
                    coverage_type=self._classify_redundancy(
                        [sections[idx] for idx in similar_indices]
                    )
                )
                redundancy_groups.append(group)
                processed.update(similar_indices)

        return redundancy_groups

    def _classify_redundancy(self,
                             sections: List[SectionRepresentation]
                            ) -> str:
        """
␣␣␣␣␣␣␣␣Determine␣if␣redundancy␣is:
␣␣␣␣␣␣␣␣-␣duplicate:␣Identical␣content
␣␣␣␣␣␣␣␣-␣complementary:␣Same␣topic,␣different␣perspectives
␣␣␣␣␣␣␣␣-␣progressive:␣Building␣on␣same␣concepts
␣␣␣␣␣␣␣␣"""
        # Compare key points
```

```
        all_points = [set(s.key_points) for s in sections]

        # Check overlap
        if len(all_points) < 2:
            return "duplicate"

        intersection = set.intersection(*all_points)
        union = set.union(*all_points)

        jaccard = len(intersection) / len(union)

        if jaccard > 0.8:
            return "duplicate"
        elif jaccard > 0.4:
            return "complementary"
        else:
            return "progressive"

@dataclass
class RedundancyGroup:
    group_id: UUID
    section_ids: List[UUID]
    similarity_scores: List[float]
    canonical_section: UUID
    coverage_type: str  # duplicate | complementary | progressive
```

8 Layer 5: Expert Reasoning and Teaching

8.1 Purpose and Epistemic Question

Epistemic Question: *"How can internal knowledge be composed into expert-level explanations and pedagogical responses?"*

Layer 5 implements the primary reasoning agent that answers user queries by dynamically selecting and composing knowledge from all abstraction levels.

8.2 Expert Reasoning Architecture

Listing 21: Expert Reasoning Agent

```
class ExpertReasoningAgent:
    def __init__(self, llm, knowledge_store):
        self.llm = llm
        self.knowledge = knowledge_store
        self.resolution_selector = ResolutionSelector()

    async def answer_query(self, query: str) -> ExpertResponse:
        """
␣␣␣␣␣␣␣␣Main␣reasoning␣loop:␣retrieve,␣synthesize,␣respond.
␣␣␣␣␣␣␣␣"""
        # Step 1: Classify query and select resolution
        resolutions = self.resolution_selector.select_optimal_level(
            query)

        # Step 2: Retrieve relevant knowledge
        retrieved = await self._multi_resolution_retrieval(
            query, resolutions
```

```
        )

        # Step 3: Check epistemic boundaries
        uncertainty = await self._assess_epistemic_uncertainty(
            query, retrieved
        )

        # Step 4: Generate response
        if uncertainty.requires_external_research:
            return await self._trigger_deep_research(query, uncertainty)
        else:
            response = await self._generate_grounded_response(
                query, retrieved, uncertainty
            )
            return response

    async def _multi_resolution_retrieval(self,
                                          query: str,
                                          resolutions: Union[int, List[int]]
                                         ) -> RetrievedKnowledge:
        """Retrieve knowledge from specified abstraction levels."""
        if isinstance(resolutions, int):
            resolutions = [resolutions]

        retrieved = RetrievedKnowledge()

        for level in resolutions:
            if level == 0:
                chunks = await self.knowledge.retrieve_raw_chunks(query)
                retrieved.raw_chunks = chunks
            elif level == 1:
                sections = await self.knowledge.retrieve_sections(query)
                retrieved.sections = sections
            elif level == 2:
                documents = await self.knowledge.retrieve_documents(query)
                retrieved.documents = documents
            elif level == 3:
                themes = await self.knowledge.retrieve_themes(query)
                retrieved.themes = themes

        return retrieved

    async def _assess_epistemic_uncertainty(self,
                                            query: str,
                                            retrieved: RetrievedKnowledge
                                           ) -> EpistemicUncertainty:
        """Determine if query can be answered with internal knowledge."""
        assessment_prompt = f"""
```

```
        Query: {query}

        Retrieved knowledge summary:
        - {len(retrieved.raw_chunks or [])} raw chunks
        - {len(retrieved.sections or [])} sections
        - {len(retrieved.documents or [])} documents
        - {len(retrieved.themes or [])} themes

        Assess:
        1. Is retrieved knowledge sufficient to answer query?
        2. Are there temporal concerns (knowledge may be outdated)?
        3. Are there conflicting claims in retrieved knowledge?
        4. What is confidence level (0-1)?
        5. Does this require external research?

        Output as JSON.
        """

        assessment = await self.llm.ainvoke(assessment_prompt)
        parsed = json.loads(assessment)

        return EpistemicUncertainty(
            is_sufficient=parsed['sufficient'],
            has_temporal_concerns=parsed['temporal_concerns'],
            has_conflicts=parsed['conflicts'],
            confidence=parsed['confidence'],
            requires_external_research=parsed['requires_research'],
            explanation=parsed.get('explanation', '')
        )

    async def _generate_grounded_response(self,
                                          query: str,
                                          retrieved: RetrievedKnowledge,
                                          uncertainty:
                                              EpistemicUncertainty
                                         ) -> ExpertResponse:
        """
        Generate response strictly from retrieved knowledge.
        """
        # Build context from retrieved knowledge
        context = self._build_context(retrieved)

        response_prompt = f"""
        You are an expert teacher with deep knowledge of the user's
        personal knowledge collection. Answer the following query using
        ONLY the provided context. Do not use external knowledge.

        Query: {query}

        Context:
        {context}

        Epistemic constraints:
        - Confidence: {uncertainty.confidence}
        - Temporal concerns: {uncertainty.has_temporal_concerns}
        - Conflicts detected: {uncertainty.has_conflicts}

        Provide:
```

```
        1. Direct answer
        2. Supporting evidence with citations
        3. Relevant prerequisite concepts if query is pedagogical
        4. Uncertainties or limitations
        5. Suggested related topics for deeper understanding

        Format response as structured JSON.
        """

        response = await self.llm.ainvoke(response_prompt)
        parsed = json.loads(response)

        return ExpertResponse(
            answer=parsed['answer'],
            citations=self._extract_citations(retrieved),
            prerequisites=parsed.get('prerequisites', []),
            uncertainties=parsed.get('uncertainties', []),
            related_topics=parsed.get('related_topics', []),
            confidence=uncertainty.confidence,
            used_resolutions=list(retrieved.get_used_levels()),
            requires_diagram=self._should_generate_diagram(query,
                parsed)
        )

@dataclass
class ExpertResponse:
    answer: str
    citations: List[Citation]
    prerequisites: List[str]
    uncertainties: List[str]
    related_topics: List[str]
    confidence: float
    used_resolutions: List[int]
    requires_diagram: bool

@dataclass
class Citation:
    source_type: str  # chunk | section | document
    source_id: UUID
    excerpt: str
    relevance_score: float
```

8.3 Pedagogical Mode

Listing 22: Pedagogical Teaching Mode

```
class PedagogicalAgent:
    def __init__(self, reasoning_agent, ontology):
        self.reasoning = reasoning_agent
        self.ontology = ontology

    async def teach_concept(self, concept: str) -> TeachingPlan:
        """
        Generate structured learning path for a concept.
        """
        # Find concept in ontology
        topic = await self.ontology.find_topic(concept)
```

```
        if not topic:
            return await self._handle_unknown_concept(concept)

        # Build prerequisite chain
        prereq_chain = await self._build_prerequisite_chain(topic)

        # Generate teaching plan
        plan = TeachingPlan(
            target_concept=concept,
            prerequisites=prereq_chain,
            learning_modules=[]
        )

        # For each level in prerequisite chain
        for level_topics in prereq_chain:
            for topic in level_topics:
                module = await self._create_learning_module(topic)
                plan.learning_modules.append(module)

        return plan

    async def _build_prerequisite_chain(self,
                                        topic: TopicNode
                                       ) -> List[List[TopicNode]]:
        """
␣␣␣␣␣␣␣␣Build␣ordered␣levels␣of␣prerequisites.
␣␣␣␣␣␣␣␣"""
        chains = []
        current_level = [topic]
        visited = {topic.topic_id}

        while current_level:
            next_level = []
            for t in current_level:
                for prereq_id in t.prerequisite_topics:
                    if prereq_id not in visited:
                        prereq = await self.ontology.get_topic(
                            prereq_id)
                        next_level.append(prereq)
                        visited.add(prereq_id)

            if next_level:
                chains.insert(0, next_level)  # Add to beginning
            current_level = next_level

        chains.append([topic])  # Target concept at end
        return chains

    async def _create_learning_module(self,
                                      topic: TopicNode
                                     ) -> LearningModule:
        """
␣␣␣␣␣␣␣␣Create␣interactive␣learning␣module␣for␣topic.
␣␣␣␣␣␣␣␣"""
        # Retrieve all content related to topic
        sections = await self.reasoning.knowledge.
            get_sections_for_topic(
```

```
            topic.topic_id
        )

        # Generate explanation
        explanation = await self.reasoning.answer_query(
            f"Explain the concept of {topic.name} in detail"
        )

        # Generate exercises
        exercises = await self._generate_exercises(topic, sections)

        return LearningModule(
            topic_name=topic.name,
            difficulty=topic.difficulty_level,
            explanation=explanation.answer,
            key_points=await self._extract_key_points(sections),
            examples=await self._extract_examples(sections),
            exercises=exercises,
            further_reading=[
                s.artifact_id for s in sections
            ]
        )

@dataclass
class LearningModule:
    topic_name: str
    difficulty: int
    explanation: str
    key_points: List[str]
    examples: List[dict]
    exercises: List[dict]
    further_reading: List[UUID]
```

9 Layer 6: Diagrammatic Explanation

9.1 Purpose and Epistemic Question

Epistemic Question: *"Can visual representation enhance understanding of this concept?"*

Layer 6 generates schematic diagrams, concept graphs, and explanatory visuals to augment textual explanations.

9.2 Diagram Generation Architecture

Listing 23: Diagrammatic Explanation Agent

```
class DiagramGenerator:
    def __init__(self, llm):
        self.llm = llm
        self.generators = {
            'concept_map': ConceptMapGenerator(),
            'flowchart': FlowchartGenerator(),
            'hierarchy': HierarchyDiagramGenerator(),
            'graph': GraphDiagramGenerator(),
            'timeline': TimelineDiagramGenerator()
        }

```

```
    async def generate_diagram(self,
                               response: ExpertResponse,
                               query: str
                              ) -> Optional[Diagram]:
        """
        Determine if diagram would be helpful and generate it.
        """
        # Assess diagram utility
        should_generate = await self._assess_diagram_utility(
            query, response
        )

        if not should_generate['useful']:
            return None

        diagram_type = should_generate['type']
        generator = self.generators.get(diagram_type)

        if not generator:
            return None

        # Extract diagram specification
        spec = await self._extract_diagram_spec(
            response, diagram_type
        )

        # Generate diagram
        diagram = await generator.generate(spec)

        return diagram

    async def _assess_diagram_utility(self,
                                      query: str,
                                      response: ExpertResponse
                                     ) -> dict:
        """
        Determine if visual representation would help.
        """
        prompt = f"""
        Query: {query}
        Response: {response.answer[:500]}...

        Would a diagram enhance understanding of this response?

        Consider:
        - Relationships between concepts
        - Hierarchical structures
        - Temporal sequences
        - Complex interactions

        If yes, suggest diagram type:
        - concept_map: Nodes and relationships
        - flowchart: Process or algorithm
        - hierarchy: Tree structure
        - graph: Network relationships
        - timeline: Temporal progression

```

```
        Output as JSON: {{"useful": true/false, "type": "...", "reason": "..."}}
        """

        assessment = await self.llm.ainvoke(prompt)
        return json.loads(assessment)

    async def _extract_diagram_spec(self,
                                    response: ExpertResponse,
                                    diagram_type: str
                                   ) -> DiagramSpec:
        """
        Extract diagram elements from response.
        """
        prompt = f"""
        Extract diagram specification from this explanation.

        Explanation: {response.answer}
        Diagram type: {diagram_type}

        Identify:
        - Nodes/entities
        - Relationships/edges
        - Labels
        - Hierarchy if applicable

        Output as JSON matching {diagram_type} schema.
        """

        spec_json = await self.llm.ainvoke(prompt)
        return DiagramSpec.from_json(spec_json, diagram_type)
```

9.3 Concept Map Generation

Listing 24: Concept Map Generator Using Graphviz

```
class ConceptMapGenerator:
    def __init__(self):
        self.graphviz = Graphviz()

    async def generate(self, spec: DiagramSpec) -> Diagram:
        """
        Generate concept map from specification.
        """
        # Create Graphviz graph
        dot = graphviz.Digraph(comment='Concept Map')
        dot.attr(rankdir='TB')
        dot.attr('node', shape='box', style='rounded,filled',
                 fillcolor='lightblue')

        # Add nodes
        for node in spec.nodes:
            dot.node(
                node['id'],
                label=self._wrap_label(node['label']),
                fillcolor=self._get_node_color(node.get('type'))
            )
```

```
        # Add edges
        for edge in spec.edges:
            dot.edge(
                edge['source'],
                edge['target'],
                label=edge.get('label', ''),
                style=self._get_edge_style(edge.get('type'))
            )

        # Render to SVG
        svg_content = dot.pipe(format='svg').decode('utf-8')

        return Diagram(
            diagram_id=uuid4(),
            diagram_type='concept_map',
            format='svg',
            content=svg_content,
            metadata={
                'node_count': len(spec.nodes),
                'edge_count': len(spec.edges)
            }
        )

    def _wrap_label(self, text: str, width: int = 20) -> str:
        """Wrap long labels for better visualization."""
        words = text.split()
        lines = []
        current_line = []
        current_length = 0

        for word in words:
            if current_length + len(word) > width:
                lines.append(' '.join(current_line))
                current_line = [word]
                current_length = len(word)
            else:
                current_line.append(word)
                current_length += len(word) + 1

        if current_line:
            lines.append(' '.join(current_line))

        return '\\n'.join(lines)

@dataclass
class Diagram:
    diagram_id: UUID
    diagram_type: str
    format: str  # svg | png | mermaid
    content: str
    metadata: dict
```

9.4 Mermaid Diagram Generation

Listing 25: Mermaid-Based Diagram Generation

```python
class FlowchartGenerator:
    async def generate(self, spec: DiagramSpec) -> Diagram:
        """
␣␣␣␣␣␣␣␣Generate␣flowchart␣using␣Mermaid␣syntax.
␣␣␣␣␣␣␣␣"""
        mermaid_code = "flowchart␣TD\\n"

        # Add nodes with styling
        for node in spec.nodes:
            node_shape = self._get_node_shape(node.get('type'))
            mermaid_code += f"␣␣␣␣{node['id']}{node_shape[0]}\"{node['
                label']}\"{node_shape[1]}\\n"

        # Add edges
        for edge in spec.edges:
            arrow = self._get_arrow_type(edge.get('type'))
            label = edge.get('label', '')
            if label:
                mermaid_code += f"␣␣␣␣{edge['source']}␣{arrow}|\"{label
                    }\"|␣{edge['target']}\\n"
            else:
                mermaid_code += f"␣␣␣␣{edge['source']}␣{arrow}␣{edge['
                    target']}\\n"

        # Add styling
        mermaid_code += self._add_styling(spec)

        return Diagram(
            diagram_id=uuid4(),
            diagram_type='flowchart',
            format='mermaid',
            content=mermaid_code,
            metadata={'mermaid_version': '9.0'}
        )

    def _get_node_shape(self, node_type: str) -> tuple:
        """Map␣node␣types␣to␣Mermaid␣shapes."""
        shapes = {
            'start': ('[', ']'),
            'end': ('([', '])'),
            'decision': ('{', '}'),
            'process': ('[', ']'),
            'data': ('[/', '/]')
        }
        return shapes.get(node_type, ('[', ']'))
```

10 Layer 7: Deep External Research Agent

10.1 Purpose and Epistemic Question

Epistemic Question: *"Does authoritative external knowledge exist that materially alters or invalidates the current internal understanding?"*

Layer 7 implements the sole deep agent in the architecture, designed to prevent epistemic stagnation through controlled external research.

10.2 DERA Architecture

Listing 26: Deep External Research Agent (DERA)

```python
class DeepExternalResearchAgent:
    def __init__(self, llm, web_search_tool, academic_apis):
        self.llm = llm
        self.web_search = web_search_tool
        self.academic = academic_apis
        self.trigger_detector = ResearchTriggerDetector()

    async def conduct_research(self,
                               query: str,
                               uncertainty: EpistemicUncertainty,
                               internal_knowledge: RetrievedKnowledge
                              ) -> ResearchReport:
        """
␣␣␣␣␣␣␣␣Conduct␣deep␣external␣research␣and␣produce␣comparative␣report.
␣␣␣␣␣␣␣␣"""
        # Step 1: Formulate research questions
        questions = await self._formulate_research_questions(
            query, uncertainty, internal_knowledge
        )

        # Step 2: Execute multi-source research
        findings = await self._execute_research(questions)

        # Step 3: Compare with internal knowledge
        comparison = await self._compare_knowledge(
            internal_knowledge, findings
        )

        # Step 4: Generate epistemic report
        report = await self._generate_report(
            query, questions, findings, comparison
        )

        return report

    async def _formulate_research_questions(self,
                                            query: str,
                                            uncertainty:
                                               EpistemicUncertainty,
                                            internal: RetrievedKnowledge
                                           ) -> List[ResearchQuestion]:
        """
␣␣␣␣␣␣␣␣Generate␣targeted␣research␣questions.
␣␣␣␣␣␣␣␣"""
        prompt = f"""
␣␣␣␣␣␣␣␣Original␣query:␣{query}

␣␣␣␣␣␣␣␣Internal␣knowledge␣summary:␣{internal.summarize()}

␣␣␣␣␣␣␣␣Epistemic␣concerns:
␣␣␣␣␣␣␣␣-␣Temporal:␣{uncertainty.has_temporal_concerns}
␣␣␣␣␣␣␣␣-␣Conflicts:␣{uncertainty.has_conflicts}
␣␣␣␣␣␣␣␣-␣Confidence:␣{uncertainty.confidence}

␣␣␣␣␣␣␣␣Generate␣specific␣research␣questions␣that␣would:
```

```
        1. Update potentially outdated information
        2. Resolve conflicting claims
        3. Fill knowledge gaps
        4. Verify critical facts

        For each question, specify:
        - Question text
        - Expected source type (academic | news | technical docs)
        - Priority (high | medium | low)

        Output as JSON list.
        """

        response = await self.llm.ainvoke(prompt)
        questions_data = json.loads(response)

        return [ResearchQuestion(**q) for q in questions_data]

    async def _execute_research(self,
                                questions: List[ResearchQuestion]
                               ) -> List[ResearchFinding]:
        """
        Execute research across multiple sources.
        """
        findings = []

        for question in questions:
            if question.expected_source == 'academic':
                # Search academic databases
                academic_results = await self.academic.search(
                    question.text,
                    filters={'since': question.get_temporal_filter()}
                )
                findings.extend(
                    self._process_academic_results(academic_results)
                )

            elif question.expected_source == 'news':
                # Search news sources
                news_results = await self.web_search.search(
                    question.text,
                    domain_filter='news',
                    recency='6months'
                )
                findings.extend(
                    self._process_news_results(news_results)
                )

            else:
                # General web search
                web_results = await self.web_search.search(
                    question.text
                )
                findings.extend(
                    self._process_web_results(web_results)
                )

        return findings
```

```
113
114     async def _compare_knowledge(self,
115                                  internal: RetrievedKnowledge,
116                                  external: List[ResearchFinding]
117                                 ) -> KnowledgeComparison:
118         """
119 ␣␣␣␣␣␣␣␣Compare␣internal␣and␣external␣knowledge.
120 ␣␣␣␣␣␣␣␣"""
121         comparison_prompt = f"""
122 ␣␣␣␣␣␣␣␣Compare␣internal␣knowledge␣with␣external␣research␣findings.
123
124 ␣␣␣␣␣␣␣␣Internal␣knowledge:
125 ␣␣␣␣␣␣␣␣{internal.detailed_summary()}
126
127 ␣␣␣␣␣␣␣␣External␣findings:
128 ␣␣␣␣␣␣␣␣{self._summarize_findings(external)}
129
130 ␣␣␣␣␣␣␣␣Identify:
131 ␣␣␣␣␣␣␣␣1.␣Confirmations:␣External␣sources␣confirm␣internal␣knowledge
132 ␣␣␣␣␣␣␣␣2.␣Updates:␣External␣sources␣provide␣more␣recent␣information
133 ␣␣␣␣␣␣␣␣3.␣Contradictions:␣External␣sources␣contradict␣internal␣
    knowledge
134 ␣␣␣␣␣␣␣␣4.␣Extensions:␣External␣sources␣add␣new␣relevant␣information
135 ␣␣␣␣␣␣␣␣5.␣Confidence␣assessment:␣How␣reliable␣are␣external␣sources?
136
137 ␣␣␣␣␣␣␣␣For␣contradictions,␣evaluate:
138 ␣␣␣␣␣␣␣␣-␣Source␣authority
139 ␣␣␣␣␣␣␣␣-␣Publication␣date
140 ␣␣␣␣␣␣␣␣-␣Evidence␣quality
141 ␣␣␣␣␣␣␣␣-␣Consensus␣vs␣outlier
142
143 ␣␣␣␣␣␣␣␣Output␣as␣structured␣JSON.
144 ␣␣␣␣␣␣␣␣"""
145
146         comparison = await self.llm.ainvoke(comparison_prompt)
147         parsed = json.loads(comparison)
148
149         return KnowledgeComparison(
150             confirmations=parsed['confirmations'],
151             updates=parsed['updates'],
152             contradictions=parsed['contradictions'],
153             extensions=parsed['extensions'],
154             confidence_assessment=parsed['confidence'],
155             recommendation=self._generate_recommendation(parsed)
156         )
157
158     async def _generate_report(self,
159                                query: str,
160                                questions: List[ResearchQuestion],
161                                findings: List[ResearchFinding],
162                                comparison: KnowledgeComparison
163                               ) -> ResearchReport:
164         """
165 ␣␣␣␣␣␣␣␣Generate␣comprehensive␣epistemic␣research␣report.
166 ␣␣␣␣␣␣␣␣"""
167         return ResearchReport(
168             report_id=uuid4(),
169             query=query,
```

```
            research_questions=questions,
            external_findings=findings,
            comparison=comparison,
            timestamp=datetime.now(),

            # Key insights
            should_update_knowledge=len(comparison.updates) > 0 or
                                    len(comparison.contradictions) > 0,
            critical_changes=comparison.get_critical_changes(),
            new_sources_to_ingest=self._identify_valuable_sources(
                findings),

            # Versioning
            affected_documents=[
                doc.document_id for doc in comparison.
                    get_affected_documents()
            ],
            proposed_updates=comparison.generate_update_proposals()
        )

@dataclass
class ResearchReport:
    report_id: UUID
    query: str
    research_questions: List[ResearchQuestion]
    external_findings: List[ResearchFinding]
    comparison: KnowledgeComparison
    timestamp: datetime
    should_update_knowledge: bool
    critical_changes: List[str]
    new_sources_to_ingest: List[str]
    affected_documents: List[UUID]
    proposed_updates: List[dict]
```

10.3 Research Trigger Detection

Listing 27: Automatic Research Trigger Detection

```
class ResearchTriggerDetector:
    def should_trigger_research(self,
                                query: str,
                                uncertainty: EpistemicUncertainty,
                                retrieved: RetrievedKnowledge
                               ) -> bool:
        """
␣␣␣␣␣␣␣␣Determine␣if␣deep␣research␣should␣be␣triggered.
␣␣␣␣␣␣␣␣"""
        triggers = []

        # Trigger 1: Explicit temporal concerns
        if uncertainty.has_temporal_concerns:
            triggers.append('temporal_concern')

        # Trigger 2: Low confidence
        if uncertainty.confidence < 0.6:
            triggers.append('low_confidence')

```

```
        # Trigger 3: Conflicting internal sources
        if uncertainty.has_conflicts:
            triggers.append('internal_conflicts')

        # Trigger 4: Sparse internal knowledge
        if self._is_sparse(retrieved):
            triggers.append('sparse_knowledge')

        # Trigger 5: Explicit request for recent information
        if self._requests_recent_info(query):
            triggers.append('explicit_recency_request')

        # Trigger 6: Known outdated topic
        if self._is_fast_evolving_topic(query):
            triggers.append('fast_evolving_topic')

        return len(triggers) > 0

    def _is_fast_evolving_topic(self, query: str) -> bool:
        """
␣␣␣␣␣␣␣␣Detect␣topics␣known␣to␣evolve␣rapidly.
␣␣␣␣␣␣␣␣"""
        fast_topics = [
            'covid', 'pandemic', 'election', 'policy',
            'regulation', 'market', 'stock', 'crypto',
            'ai␣model', 'language␣model', 'gpt'
        ]
        return any(topic in query.lower() for topic in fast_topics)
```

10.4 Knowledge Versioning

Listing 28: Versioned Knowledge Updates

```
class KnowledgeVersionManager:
    def __init__(self, db):
        self.db = db

    async def apply_research_updates(self,
                                     report: ResearchReport):
        """
␣␣␣␣␣␣␣␣Apply␣research␣findings␣as␣versioned␣updates.
␣␣␣␣␣␣␣␣"""
        for update in report.proposed_updates:
            # Create new version
            version = KnowledgeVersion(
                version_id=uuid4(),
                target_id=update['target_id'],
                target_type=update['target_type'],
                change_type=update['change_type'],
                old_content=await self._get_current_content(
                    update['target_id']
                ),
                new_content=update['new_content'],
                source_report=report.report_id,
                timestamp=datetime.now(),
                validated=False
            )
```

```
            await self.db.save_version(version)

        # Mark report as processed
        await self.db.mark_report_processed(report.report_id)

    async def get_version_history(self,
                                  entity_id: UUID
                                 ) -> List[KnowledgeVersion]:
        """
        Retrieve version history for any knowledge entity.
        """
        return await self.db.query_versions(entity_id)

@dataclass
class KnowledgeVersion:
    version_id: UUID
    target_id: UUID
    target_type: str
    change_type: str  # update | contradiction | extension
    old_content: dict
    new_content: dict
    source_report: UUID
    timestamp: datetime
    validated: bool
```

11 User Interaction Layer: CLI Interface

11.1 CLI Architecture

The user interaction layer is built on the `llm` framework by Simon Willison, providing a scriptable, composable command-line interface.

Listing 29: KNOWY CLI Plugin Implementation

```
import llm
from typing import Optional

@llm.hookimpl
def register_commands(cli):
    @cli.group()
    def knowy():
        """KNOWY personal knowledge system commands."""
        pass

    @knowy.command()
    @click.argument("reference")
    @click.option("--origin", default="manual")
    def ingest(reference, origin):
        """Ingest a new knowledge reference."""
        ingestion_service = IngestionService()
        result = asyncio.run(
            ingestion_service.ingest(reference, origin)
        )
        click.echo(f"Ingested: {result.source_id}")

    @knowy.command()
```

```
    @click.argument("query")
    @click.option("--mode", type=click.Choice(['expert', 'teach']),
                  default='expert')
    @click.option("--with-diagram", is_flag=True)
    def ask(query, mode, with_diagram):
        """Query the knowledge system."""
        if mode == 'expert':
            agent = ExpertReasoningAgent()
            response = asyncio.run(agent.answer_query(query))
        else:
            agent = PedagogicalAgent()
            response = asyncio.run(agent.teach_concept(query))

        click.echo(response.answer)

        if with_diagram and response.requires_diagram:
            diagram_gen = DiagramGenerator()
            diagram = asyncio.run(
                diagram_gen.generate_diagram(response, query)
            )
            if diagram:
                save_diagram(diagram)

    @knowy.command()
    @click.argument("topic")
    def topics(topic):
        """Explore topic ontology."""
        ontology = OntologyService()
        topic_info = asyncio.run(ontology.get_topic_info(topic))

        click.echo(f"Topic: {topic_info.name}")
        click.echo(f"Description: {topic_info.description}")
        click.echo(f"
Prerequisites:")
        for prereq in topic_info.prerequisite_topics:
            click.echo(f"  - {prereq.name}")

    @knowy.command()
    @click.option("--since", type=click.DateTime())
    def research_reports(since):
        """View deep research reports."""
        research_service = ResearchService()
        reports = asyncio.run(
            research_service.get_reports(since=since)
        )

        for report in reports:
            click.echo(f"
Report: {report.report_id}")
            click.echo(f"Query: {report.query}")
            click.echo(f"Findings: {len(report.external_findings)}")
            click.echo(f"Updates proposed: {len(report.proposed_updates
                )}")
```

11.2 CLI Usage Examples

Listing 30: CLI Usage Examples

```
# Ingest a paper from URL
$ llm knowy ingest "https://arxiv.org/abs/2103.00020" --origin browser

# Ingest a YouTube course
$ llm knowy ingest "https://youtube.com/playlist?list=..." --origin
    manual

# Ask a question
$ llm knowy ask "Explain the transformer architecture"

# Get pedagogical explanation with diagram
$ llm knowy ask "How does backpropagation work?" --mode teach --with-
    diagram

# Explore topics
$ llm knowy topics "machine learning"

# View research reports
$ llm knowy research-reports --since "2026-01-01"

# Scriptable workflows
$ cat papers.txt | while read url; do
    llm knowy ingest "$url"
done

# Export knowledge to file
$ llm knowy ask "Summarize all papers on attention mechanisms" >
    summary.md
```

12 System Integration and Orchestration

12.1 LangGraph Orchestration

Listing 31: LangGraph Workflow Orchestration

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator

class KnowyState(TypedDict):
    """Global state for KNOWY orchestration."""
    # Input
    user_query: str
    mode: str  # ingest | query | teach | research

    # Ingestion flow
    raw_input: Optional[str]
    source_descriptor: Optional[CanonicalSourceDescriptor]
    artifacts: Optional[List[KnowledgeArtifact]]
    acquired_content: Optional[List[Any]]

    # Query flow
    retrieved_knowledge: Optional[RetrievedKnowledge]
    epistemic_uncertainty: Optional[EpistemicUncertainty]
    expert_response: Optional[ExpertResponse]
```

```
    diagram: Optional[Diagram]

    # Research flow
    research_triggered: bool
    research_report: Optional[ResearchReport]

    # Output
    final_response: str
    metadata: dict

def create_knowy_workflow() -> StateGraph:
    """
␣␣␣␣Create␣the␣complete␣KNOWY␣orchestration␣graph.
␣␣␣␣"""
    workflow = StateGraph(KnowyState)

    # Add nodes for each layer
    workflow.add_node("route_mode", route_by_mode)
    workflow.add_node("ingest_l0", layer0_ingestion)
    workflow.add_node("expand_l1", layer1_expansion)
    workflow.add_node("acquire_l2", layer2_acquisition)
    workflow.add_node("represent_l3", layer3_representation)
    workflow.add_node("classify_l4", layer4_classification)
    workflow.add_node("reason_l5", layer5_reasoning)
    workflow.add_node("diagram_l6", layer6_diagramming)
    workflow.add_node("research_l7", layer7_deep_research)
    workflow.add_node("finalize", finalize_response)

    # Define edges
    workflow.set_entry_point("route_mode")

    # Routing logic
    workflow.add_conditional_edges(
        "route_mode",
        determine_flow,
        {
            "ingest": "ingest_l0",
            "query": "reason_l5",
            "teach": "reason_l5",
            "research": "research_l7"
        }
    )

    # Ingestion flow
    workflow.add_edge("ingest_l0", "expand_l1")
    workflow.add_edge("expand_l1", "acquire_l2")
    workflow.add_edge("acquire_l2", "represent_l3")
    workflow.add_edge("represent_l3", "classify_l4")
    workflow.add_edge("classify_l4", "finalize")

    # Query flow
    workflow.add_conditional_edges(
        "reason_l5",
        check_research_needed,
        {
            "research": "research_l7",
            "diagram": "diagram_l6",
            "finalize": "finalize"
```

```
        }
    )

    workflow.add_edge("diagram_16", "finalize")
    workflow.add_edge("research_17", "reason_15")  # Loop back
    workflow.add_edge("finalize", END)

    return workflow.compile()

def determine_flow(state: KnowyState) -> str:
    """Route to appropriate flow based on mode."""
    return state["mode"]

def check_research_needed(state: KnowyState) -> str:
    """Determine next step after reasoning."""
    if state.get("research_triggered"):
        return "research"
    elif state.get("expert_response", {}).get("requires_diagram"):
        return "diagram"
    else:
        return "finalize"
```

12.2 Deployment Architecture

Listing 32: Complete Deployment Configuration

```
# docker-compose.yml equivalent in Python config

@dataclass
class DeploymentConfig:
    """KNOWY deployment configuration."""

    # Local LLM configuration
    ollama_host: str = "localhost"
    ollama_port: int = 11434
    primary_model: str = "llama3.2"
    vision_model: str = "llama3.2-vision"

    # Database configuration
    postgres_host: str = "localhost"
    postgres_port: int = 5432
    postgres_db: str = "knowy_knowledge"
    neo4j_host: str = "localhost"
    neo4j_port: int = 7687

    # Vector database
    vector_db_type: str = "qdrant"  # or "faiss"
    qdrant_host: str = "localhost"
    qdrant_port: int = 6333

    # Storage paths
    data_dir: Path = Path.home() / ".knowy" / "data"
    cache_dir: Path = Path.home() / ".knowy" / "cache"
    models_dir: Path = Path.home() / ".knowy" / "models"

    # Processing configuration
    max_concurrent_acquisitions: int = 5
```

```
    chunk_size: int = 512
    embedding_dim: int = 1536

    # Research configuration
    enable_deep_research: bool = True
    research_confidence_threshold: float = 0.6
    max_research_sources: int = 20

def initialize_system(config: DeploymentConfig):
    """Initialize KNOWY system with configuration."""
    # Create directories
    config.data_dir.mkdir(parents=True, exist_ok=True)
    config.cache_dir.mkdir(parents=True, exist_ok=True)

    # Initialize databases
    db_manager = DatabaseManager(config)
    db_manager.initialize_schemas()

    # Initialize Ollama connection
    ollama_client = OllamaClient(
        host=config.ollama_host,
        port=config.ollama_port
    )

    # Pull required models
    ollama_client.pull(config.primary_model)
    ollama_client.pull(config.vision_model)

    # Initialize vector database
    if config.vector_db_type == "qdrant":
        vector_db = QdrantClient(
            host=config.qdrant_host,
            port=config.qdrant_port
        )
        vector_db.create_collection(
            collection_name="knowy_chunks",
            vectors_config=VectorParams(
                size=config.embedding_dim,
                distance=Distance.COSINE
            )
        )

    # Initialize services
    services = ServiceRegistry(
        db_manager=db_manager,
        llm_client=ollama_client,
        vector_db=vector_db,
        config=config
    )

    return services
```

13 Performance Optimization

13.1 Caching Strategy

Listing 33: Multi-Level Caching Implementation

```python
class CacheManager:
    def __init__(self, config: DeploymentConfig):
        self.config = config
        self.embedding_cache = LRUCache(maxsize=10000)
        self.query_cache = TTLCache(maxsize=1000, ttl=3600)

    async def get_or_compute_embedding(self,
                                       text: str,
                                       model: str
                                      ) -> np.ndarray:
        """Cache embeddings to avoid recomputation."""
        cache_key = hash(text + model)

        if cache_key in self.embedding_cache:
            return self.embedding_cache[cache_key]

        embedding = await self._compute_embedding(text, model)
        self.embedding_cache[cache_key] = embedding
        return embedding

    async def get_or_execute_query(self,
                                   query: str,
                                   mode: str
                                  ) -> Any:
        """Cache query results with TTL."""
        cache_key = hash(query + mode)

        if cache_key in self.query_cache:
            return self.query_cache[cache_key]

        result = await self._execute_query(query, mode)
        self.query_cache[cache_key] = result
        return result
```

13.2 Parallel Processing

Listing 34: Parallel Acquisition Processing

```python
async def parallel_acquisition(artifacts: List[KnowledgeArtifact],
                               max_concurrent: int = 5
                              ) -> List[AcquiredContent]:
    """Process multiple artifacts concurrently."""
    semaphore = asyncio.Semaphore(max_concurrent)

    async def acquire_with_limit(artifact):
        async with semaphore:
            return await acquire_artifact(artifact)

    tasks = [acquire_with_limit(a) for a in artifacts]
    results = await asyncio.gather(*tasks, return_exceptions=True)

    # Filter out exceptions and log failures
    successful = []
    for artifact, result in zip(artifacts, results):
        if isinstance(result, Exception):
```

```
            logger.error(f"Failed to acquire {artifact.artifact_id}: {result}")
        else:
            successful.append(result)

    return successful
```

14 Evaluation and Metrics

14.1 Epistemic Integrity Metrics

Listing 35: Epistemic Quality Metrics

```
@dataclass
class EpistemicMetrics:
    """Metrics for evaluating epistemic integrity."""

    # Hallucination detection
    hallucination_rate: float
    unsupported_claims: int
    total_claims: int

    # Citation quality
    citation_precision: float  # Relevant citations / total citations
    citation_coverage: float   # Claims with citations / total claims

    # Uncertainty quantification
    avg_confidence: float
    confidence_calibration: float  # How well confidence matches accuracy

    # Knowledge freshness
    avg_document_age_days: float
    outdated_documents: int
    research_trigger_rate: float

    def compute_epistemic_score(self) -> float:
        """Composite score for epistemic quality."""
        return (
            (1 - self.hallucination_rate) * 0.3 +
            self.citation_precision * 0.2 +
            self.citation_coverage * 0.2 +
            self.confidence_calibration * 0.15 +
            min(1.0, 365 / max(self.avg_document_age_days, 1)) * 0.15
        )

class MetricsCollector:
    def __init__(self):
        self.metrics_db = MetricsDatabase()

    async def evaluate_response(self,
                                query: str,
                                response: ExpertResponse,
                                retrieved: RetrievedKnowledge
                               ) -> EpistemicMetrics:
        """Evaluate a single response."""
        # Detect hallucinations
```

```
        hallucination_check = await self._check_hallucinations(
            response.answer, retrieved
        )

        # Evaluate citations
        citation_metrics = self._evaluate_citations(
            response.citations, retrieved
        )

        # Check freshness
        freshness = self._compute_freshness(retrieved)

        return EpistemicMetrics(
            hallucination_rate=hallucination_check.rate,
            unsupported_claims=hallucination_check.unsupported,
            total_claims=hallucination_check.total,
            citation_precision=citation_metrics.precision,
            citation_coverage=citation_metrics.coverage,
            avg_confidence=response.confidence,
            confidence_calibration=0.0,  # Requires ground truth
            avg_document_age_days=freshness.avg_age,
            outdated_documents=freshness.outdated_count,
            research_trigger_rate=0.0  # Computed over time
        )
```

14.2 Performance Metrics

Listing 36: System Performance Monitoring

```
@dataclass
class PerformanceMetrics:
    """Performance and efficiency metrics."""

    # Latency
    avg_query_latency_ms: float
    p95_query_latency_ms: float
    p99_query_latency_ms: float

    # Throughput
    queries_per_minute: float
    ingestions_per_hour: float

    # Resource utilization
    avg_memory_usage_gb: float
    peak_memory_usage_gb: float
    avg_cpu_percent: float

    # Storage
    total_chunks: int
    total_documents: int
    total_size_gb: float

    # Cache efficiency
    embedding_cache_hit_rate: float
    query_cache_hit_rate: float

class PerformanceMonitor:
```

```
    def __init__(self):
        self.start_time = time.time()
        self.query_latencies = []

    @contextmanager
    def measure_latency(self):
        """Context manager for latency measurement."""
        start = time.time()
        yield
        latency = (time.time() - start) * 1000
        self.query_latencies.append(latency)

    def get_metrics(self) -> PerformanceMetrics:
        """Compute current performance metrics."""
        return PerformanceMetrics(
            avg_query_latency_ms=np.mean(self.query_latencies),
            p95_query_latency_ms=np.percentile(self.query_latencies,
                95),
            p99_query_latency_ms=np.percentile(self.query_latencies,
                99),
            queries_per_minute=len(self.query_latencies) /
                               ((time.time() - self.start_time) / 60),
            # ... other metrics
        )
```

15 Testing Strategy

15.1 Unit Testing

Listing 37: Layer Testing Examples

```
import pytest

class TestLayer0Ingestion:
    @pytest.fixture
    def ingestion_agent(self):
        return IngestionAgent()

    @pytest.mark.asyncio
    async def test_pdf_detection(self, ingestion_agent):
        """Test PDF format detection."""
        url = "https://arxiv.org/pdf/2103.00020.pdf"
        descriptor = await ingestion_agent.create_descriptor(url, "
            manual")

        assert descriptor.content_hints.contains_pdf is True
        assert descriptor.content_hints.contains_video is False

    @pytest.mark.asyncio
    async def test_youtube_detection(self, ingestion_agent):
        """Test YouTube video detection."""
        url = "https://www.youtube.com/watch?v=abc123"
        descriptor = await ingestion_agent.create_descriptor(url, "
            browser")

        assert descriptor.content_hints.contains_video is True
        assert descriptor.links == [url]
```

```python

class TestLayer3Representation:
    @pytest.fixture
    def summarizer(self):
        return SectionSummarizer(llm=MockLLM())

    @pytest.mark.asyncio
    async def test_section_summary_generation(self, summarizer):
        """Test section summary creation."""
        chunks = create_mock_chunks(count=5)
        summary = await summarizer.generate_section_summary(chunks)

        assert summary.main_claim is not None
        assert len(summary.key_points) > 0
        assert len(summary.chunk_ids) == 5

class TestLayer7Research:
    @pytest.mark.asyncio
    async def test_research_trigger_temporal(self):
        """Test research trigger on temporal concerns."""
        detector = ResearchTriggerDetector()
        uncertainty = EpistemicUncertainty(
            has_temporal_concerns=True,
            confidence=0.8
        )

        should_trigger = detector.should_trigger_research(
            "What is the current policy?",
            uncertainty,
            MockRetrievedKnowledge()
        )

        assert should_trigger is True
```

15.2 Integration Testing

Listing 38: End-to-End Integration Tests

```python
class TestEndToEnd:
    @pytest.mark.asyncio
    @pytest.mark.slow
    async def test_complete_ingestion_pipeline(self):
        """Test complete ingestion from URL to representation."""
        system = await initialize_test_system()

        # Ingest a paper
        result = await system.ingest(
            "https://arxiv.org/abs/1706.03762",  # Attention Is All You Need
            origin="test"
        )

        # Verify all layers executed
        assert result.source_descriptor is not None
        assert len(result.artifacts) > 0
        assert len(result.sections) > 0
        assert result.document_representation is not None
```

```
        # Verify queryable
        response = await system.query(
            "Explain the transformer architecture"
        )
        assert "attention" in response.answer.lower()
        assert len(response.citations) > 0

    @pytest.mark.asyncio
    async def test_research_integration(self):
        """Test deep research integration."""
        system = await initialize_test_system()

        # Create scenario requiring research
        response = await system.query(
            "What are the latest developments in LLMs?",
            force_research=True
        )

        assert response.research_report is not None
        assert len(response.research_report.external_findings) > 0
```

16 Security and Privacy

16.1 Data Privacy

Listing 39: Privacy-Preserving Design

```
class PrivacyManager:
    """Ensure local-first privacy guarantees."""

    def __init__(self, config: DeploymentConfig):
        self.config = config
        self.encryption_key = self._load_or_generate_key()

    def encrypt_sensitive_content(self, content: str) -> bytes:
        """Encrypt sensitive content before storage."""
        from cryptography.fernet import Fernet
        f = Fernet(self.encryption_key)
        return f.encrypt(content.encode())

    def anonymize_for_research(self,
                               query: str
                              ) -> str:
        """Remove PII before external research."""
        # Remove email addresses
        query = re.sub(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z
            ]{2,}\b',
                       '[EMAIL]', query)

        # Remove phone numbers
        query = re.sub(r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b',
                       '[PHONE]', query)

        # Remove names (simple heuristic)
        query = self._remove_proper_names(query)

```

```
        return query

    def validate_no_data_leakage(self,
                                 response: ExpertResponse
                                ) -> bool:
        """Ensure response doesn't leak private info."""
        # Check for common PII patterns
        has_email = '@' in response.answer
        has_phone = re.search(r'\d{3}[-.]?\d{3}[-.]?\d{4}', response.
            answer)

        return not (has_email or has_phone)
```

17 Future Extensions

17.1 Planned Enhancements

Enhancement	Description
Collaborative Knowledge	Multi-user knowledge sharing with access control
Active Learning	System proposes questions to fill knowledge gaps
Spaced Repetition	Integration with learning science for retention
Voice Interface	Speech-to-text query input and text-to-speech output
Mobile App	iOS/Android clients with offline capability
Browser Copilot	Real-time knowledge assistance while browsing
Export Formats	Export knowledge as books, courses, documentation
Graph Visualization	Interactive exploration of knowledge graph

Table 5: Planned Future Enhancements

18 Conclusion

KNOWY represents a comprehensive solution to the challenge of personal knowledge management in the age of AI. By enforcing strict epistemic boundaries, implementing multi-resolution knowledge representation, and integrating controlled external research, the system achieves:

- **Epistemic Integrity**: Hallucination minimization through grounded reasoning and uncertainty quantification
- **Multimodal Understanding**: Native support for video, diagrams, and text with semantic alignment
- **Knowledge Evolution**: Principled external research without epistemic drift
- **Expert Reasoning**: Multi-level abstraction enabling both fact lookup and conceptual understanding

- **Pedagogical Capability**: Structured teaching with prerequisite chains and learning modules
- **Local-First Design**: Privacy preservation and workflow composability through CLI

The architecture is production-ready and provides a robust foundation for future enhancements in collaborative knowledge sharing, active learning, and advanced user interfaces.

A Appendix A: Database Schemas

Listing 40: Complete PostgreSQL Schema

```
-- Ingestion layer
CREATE TABLE ingestion_sources (
    source_id UUID PRIMARY KEY,
    origin VARCHAR(50) NOT NULL,
    raw_reference TEXT NOT NULL,
    raw_text TEXT,
    links JSONB,
    content_hints JSONB NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Knowledge artifacts
CREATE TABLE knowledge_artifacts (
    artifact_id UUID PRIMARY KEY,
    source_id UUID REFERENCES ingestion_sources(source_id),
    artifact_type VARCHAR(50) NOT NULL,
    title TEXT NOT NULL,
    authors JSONB,
    url TEXT,
    doi TEXT,
    isbn TEXT,
    publication_date TIMESTAMPTZ,
    metadata JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Raw chunks (Level 0)
CREATE TABLE raw_chunks (
    chunk_id UUID PRIMARY KEY,
    artifact_id UUID REFERENCES knowledge_artifacts(artifact_id),
    chunk_type VARCHAR(50),
    text_content TEXT,
    visual_content BYTEA,
    structured_content JSONB,
    position_in_artifact INTEGER,
    section_id UUID,
    embedding vector(1536),
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Section representations (Level 1)
CREATE TABLE section_representations (
    section_id UUID PRIMARY KEY,
    artifact_id UUID REFERENCES knowledge_artifacts(artifact_id),
```

```
    title TEXT NOT NULL,
    main_claim TEXT,
    key_points JSONB,
    terminology JSONB,
    chunk_ids JSONB,
    embedding vector(1536),
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Document representations (Level 2)
CREATE TABLE document_representations (
    document_id UUID PRIMARY KEY,
    artifact_id UUID REFERENCES knowledge_artifacts(artifact_id),
    central_thesis TEXT,
    contributions JSONB,
    methodology TEXT,
    findings JSONB,
    novel_concepts JSONB,
    embedding vector(1536),
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Thematic synthesis (Level 3)
CREATE TABLE thematic_syntheses (
    theme_id UUID PRIMARY KEY,
    theme_name TEXT NOT NULL,
    document_ids JSONB,
    agreements JSONB,
    disagreements JSONB,
    key_terminology JSONB,
    embedding vector(1536),
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Research reports
CREATE TABLE research_reports (
    report_id UUID PRIMARY KEY,
    query TEXT NOT NULL,
    external_findings JSONB,
    comparison JSONB,
    should_update_knowledge BOOLEAN,
    affected_documents JSONB,
    proposed_updates JSONB,
    timestamp TIMESTAMPTZ DEFAULT NOW()
);

-- Knowledge versions
CREATE TABLE knowledge_versions (
    version_id UUID PRIMARY KEY,
    target_id UUID NOT NULL,
    target_type VARCHAR(50),
    change_type VARCHAR(50),
    old_content JSONB,
    new_content JSONB,
    source_report UUID REFERENCES research_reports(report_id),
    validated BOOLEAN DEFAULT FALSE,
    timestamp TIMESTAMPTZ DEFAULT NOW()
);
```

```
104
105 -- Indexes
106 CREATE INDEX idx_artifacts_type ON knowledge_artifacts(artifact_type);
107 CREATE INDEX idx_chunks_artifact ON raw_chunks(artifact_id);
108 CREATE INDEX idx_chunks_embedding ON raw_chunks
109     USING ivfflat (embedding vector_cosine_ops);
110 CREATE INDEX idx_sections_artifact ON section_representations(
    artifact_id);
111 CREATE INDEX idx_documents_artifact ON document_representations(
    artifact_id);
```

B Appendix B: Configuration Files

Listing 41: System Configuration (config.yaml)

```
1  # KNOWY Configuration File
2
3  system:
4    name: "KNOWY"
5    version: "1.0.0"
6    data_dir: "~/.knowy/data"
7    cache_dir: "~/.knowy/cache"
8    log_level: "INFO"
9
10 llm:
11   provider: "ollama"
12   host: "localhost"
13   port: 11434
14   models:
15     primary: "llama3.2:latest"
16     vision: "llama3.2-vision:latest"
17     embedding: "nomic-embed-text:latest"
18
19 databases:
20   postgres:
21     host: "localhost"
22     port: 5432
23     database: "knowy_knowledge"
24     user: "knowy"
25     password: "${POSTGRES_PASSWORD}"
26
27   neo4j:
28     host: "localhost"
29     port: 7687
30     user: "neo4j"
31     password: "${NEO4J_PASSWORD}"
32
33   qdrant:
34     host: "localhost"
35     port: 6333
36     collection_name: "knowy_chunks"
37
38 processing:
39   max_concurrent_acquisitions: 5
40   chunk_size: 512
41   chunk_overlap: 50
```

```
  embedding_batch_size: 32

video_analysis:
  frame_extraction_fps: 1
  shot_detection_threshold: 0.3
  max_frames_per_shot: 10

research:
  enabled: true
  confidence_threshold: 0.6
  max_sources: 20
  academic_api_key: "${SEMANTIC_SCHOLAR_API_KEY}"

cache:
  embedding_cache_size: 10000
  query_cache_size: 1000
  query_cache_ttl_seconds: 3600
```

C Appendix C: API Reference

Listing 42: Public API Reference

```
from knowy import KNOWY

# Initialize system
system = KNOWY.initialize(config_path="config.yaml")

# Ingestion API
system.ingest(
    reference: str,
    origin: str = "manual",
    metadata: dict = None
) -> IngestionResult

# Query API
system.query(
    query: str,
    mode: str = "expert",  # expert | teach
    with_diagram: bool = False,
    force_research: bool = False
) -> QueryResult

# Teaching API
system.teach(
    concept: str,
    difficulty: int = None  # Auto-detect if None
) -> TeachingPlan

# Ontology API
system.ontology.get_topic(topic_name: str) -> TopicNode
system.ontology.get_prerequisites(topic: TopicNode) -> List[TopicNode]
system.ontology.list_topics() -> List[TopicNode]

# Research API
system.research.get_reports(
    since: datetime = None,
    limit: int = 10
```

```
) -> List[ResearchReport]

system.research.apply_updates(
    report_id: UUID,
    validate: bool = True
) -> None

# Metrics API
system.metrics.get_epistemic_metrics() -> EpistemicMetrics
system.metrics.get_performance_metrics() -> PerformanceMetrics

# Export API
system.export.to_markdown(
    topic: str = None,
    output_path: str
) -> None

system.export.to_pdf(
    topic: str = None,
    output_path: str
) -> None
```

References

1. LangGraph: https://github.com/langchain-ai/langgraph
2. Ollama: https://github.com/ollama/ollama
3. LLaMA 3: https://ai.meta.com/llama/
4. video-analyzer: https://github.com/byjlw/video-analyzer
5. llm CLI: https://github.com/simonw/llm
6. FAISS: https://github.com/facebookresearch/faiss
7. Qdrant: https://qdrant.tech
8. Deep Agents Paper: https://huggingface.co/papers/2510.21618